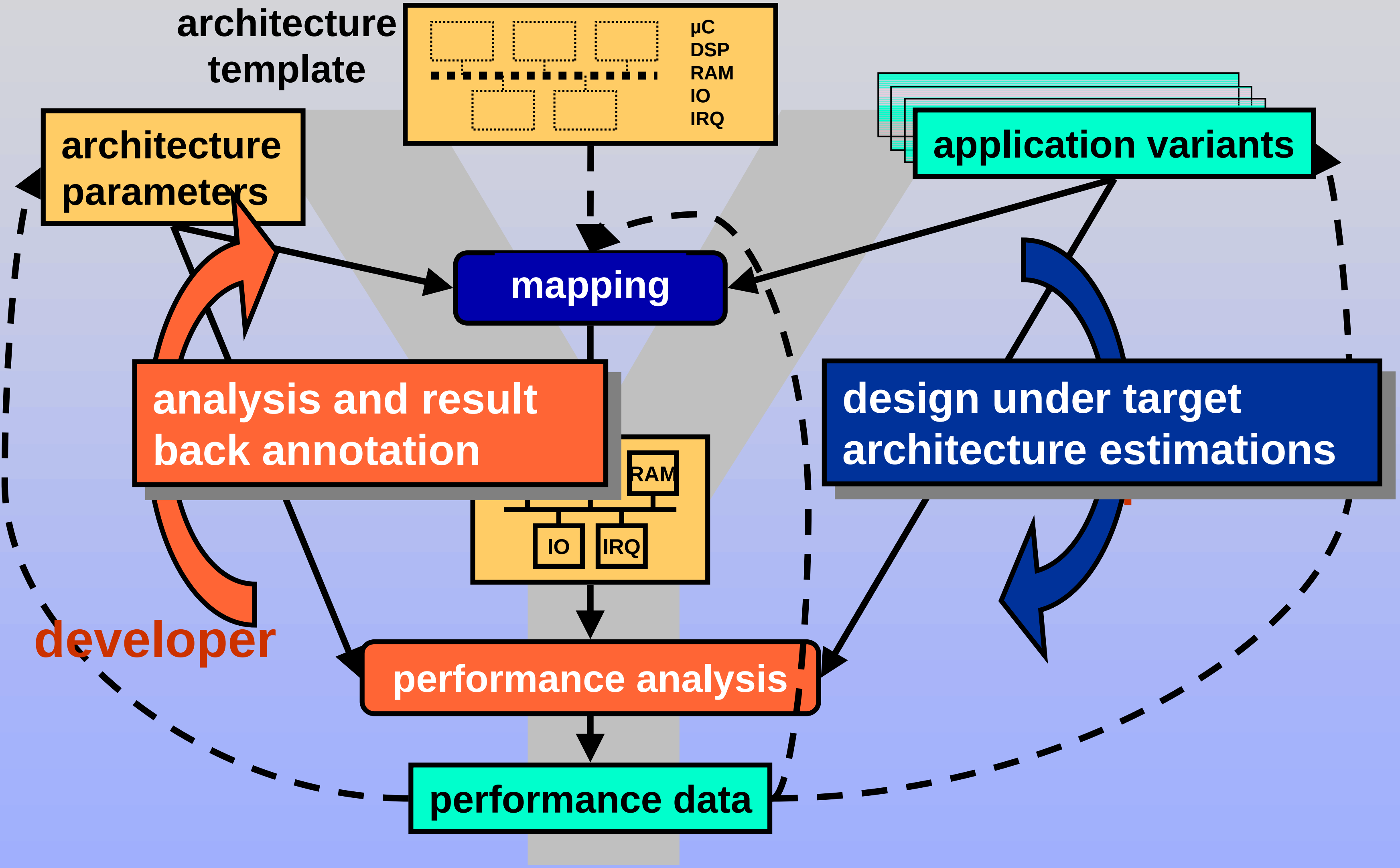
MPSoC Platform Performance Modeling

R. Ernst

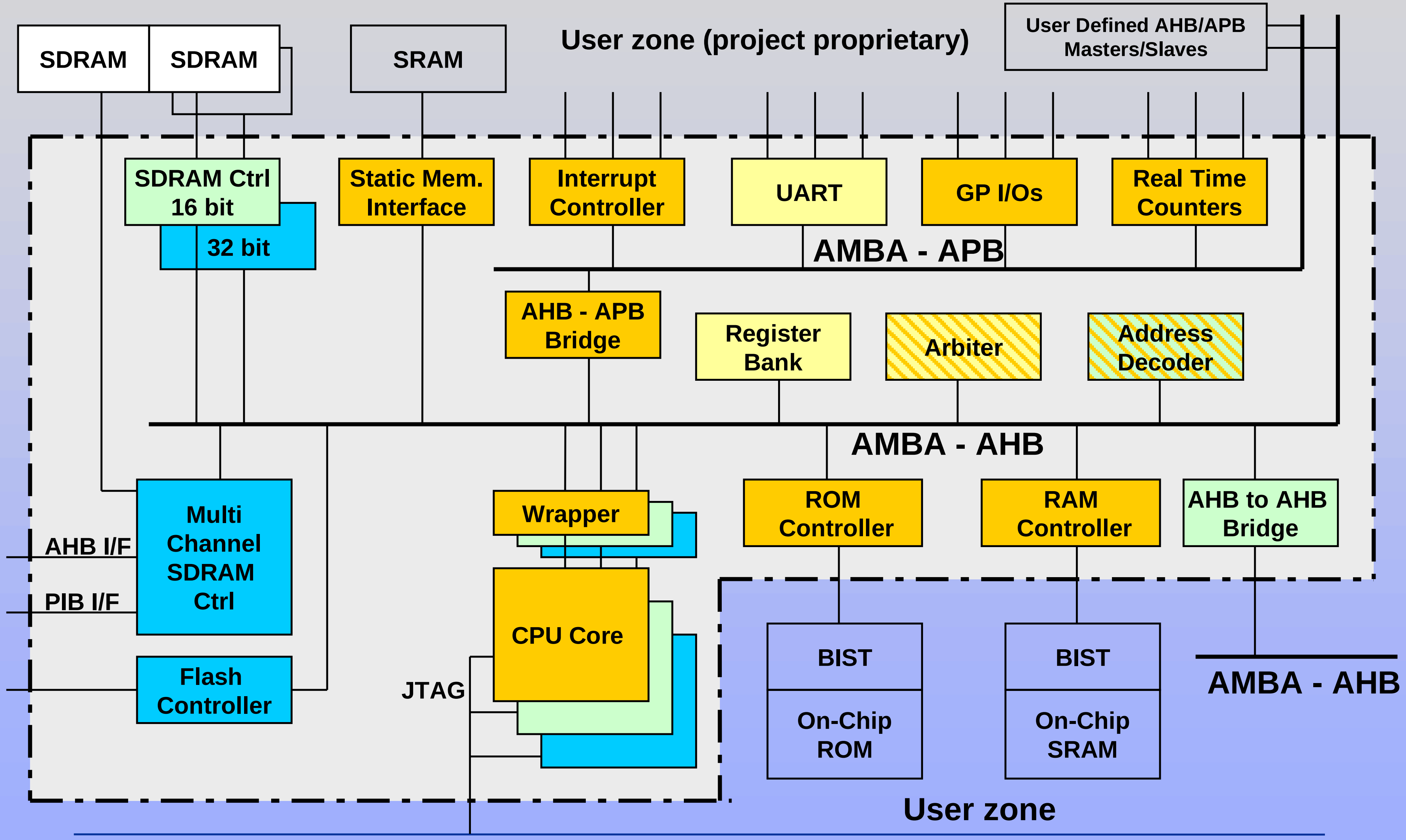
TU Braunschweig

Overview

The role of performance modeling



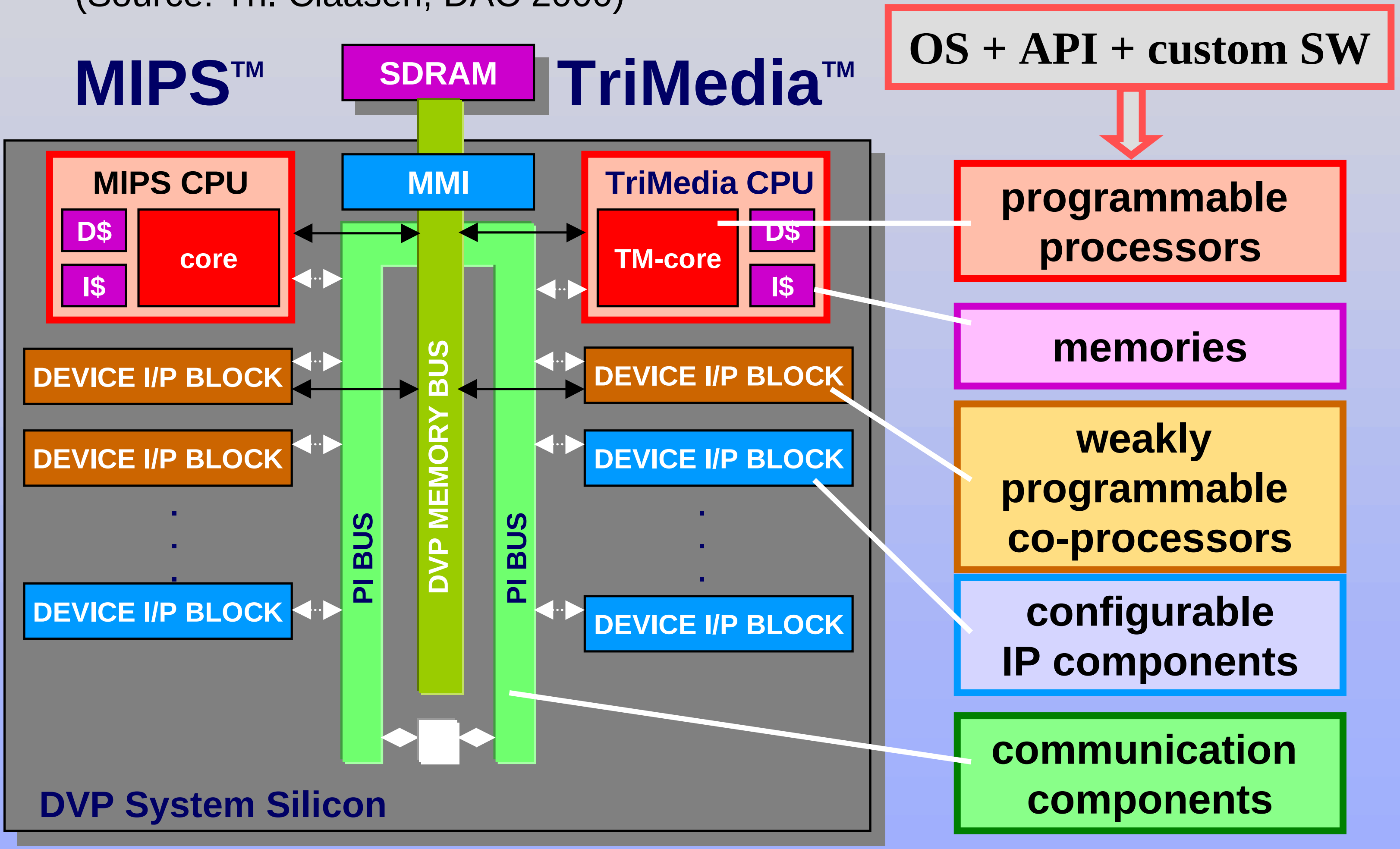
Platform HW complexity



Source: M. Münch, Alcatel

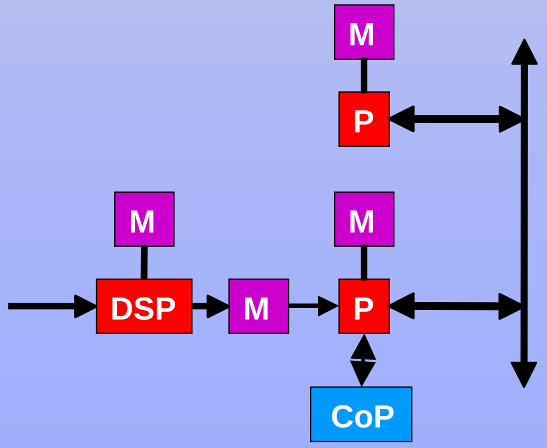
Platform component types

- **Another example: Philips Nexperia™ platform**
  (Source: Th. Claasen, DAC 2000)



Platform architectures are heterogeneous

- different processing element types
  - processors, weakly programmable coprocessors, IP components
- different interconnection networks and communication protocols
- different memory types
- different scheduling and synchronization strategies

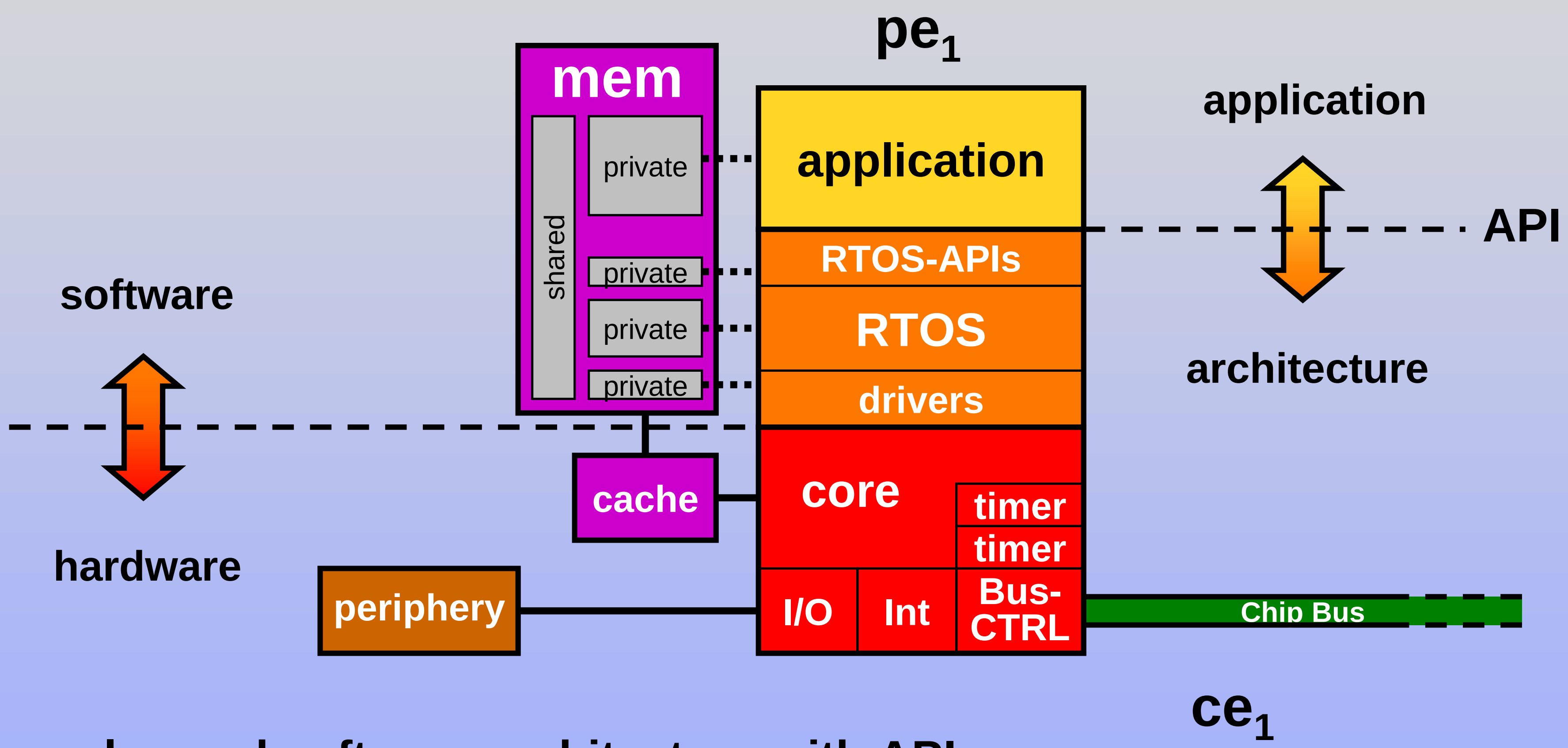


Managing HW platform complexity

- development of APIs to hide complexity from application programmer and improve portability
- specialized RTOS to control resource sharing and interfaces

⇒ complex multi-level HW/SW architecture

Software architecture example



- layered software architecture with API

⇒ SW is heterogeneous

Platform design challenges

- integration
  - design process integration
  - heterogeneous component and language integration (VSIA, Accellera)
- design space exploration and optimization
- verification

Platform verification

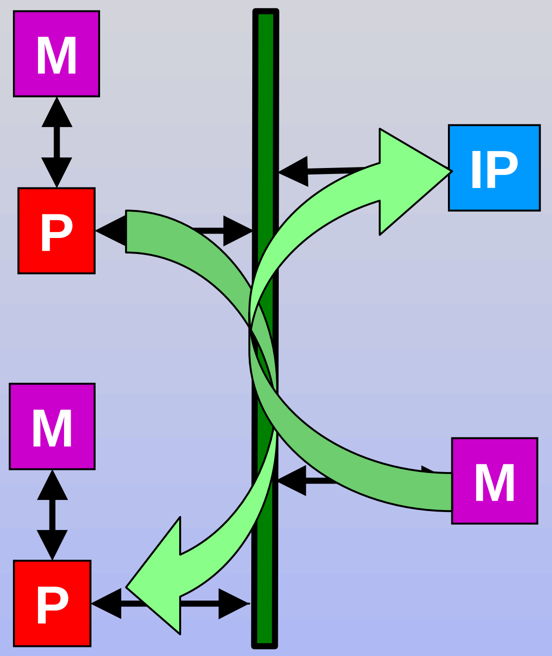
- correct implementation of specified function
  - HW/SW co-simulation (CVE, CoWare, CoCentric, VCC), verification

} general design problem

- correct target architecture parameters
  - processor and communication performance
  - adherence to timing requirements
  - no memory over/underflow
  - no run-time dependent dead-locks

} challenge to heterogeneous platform design

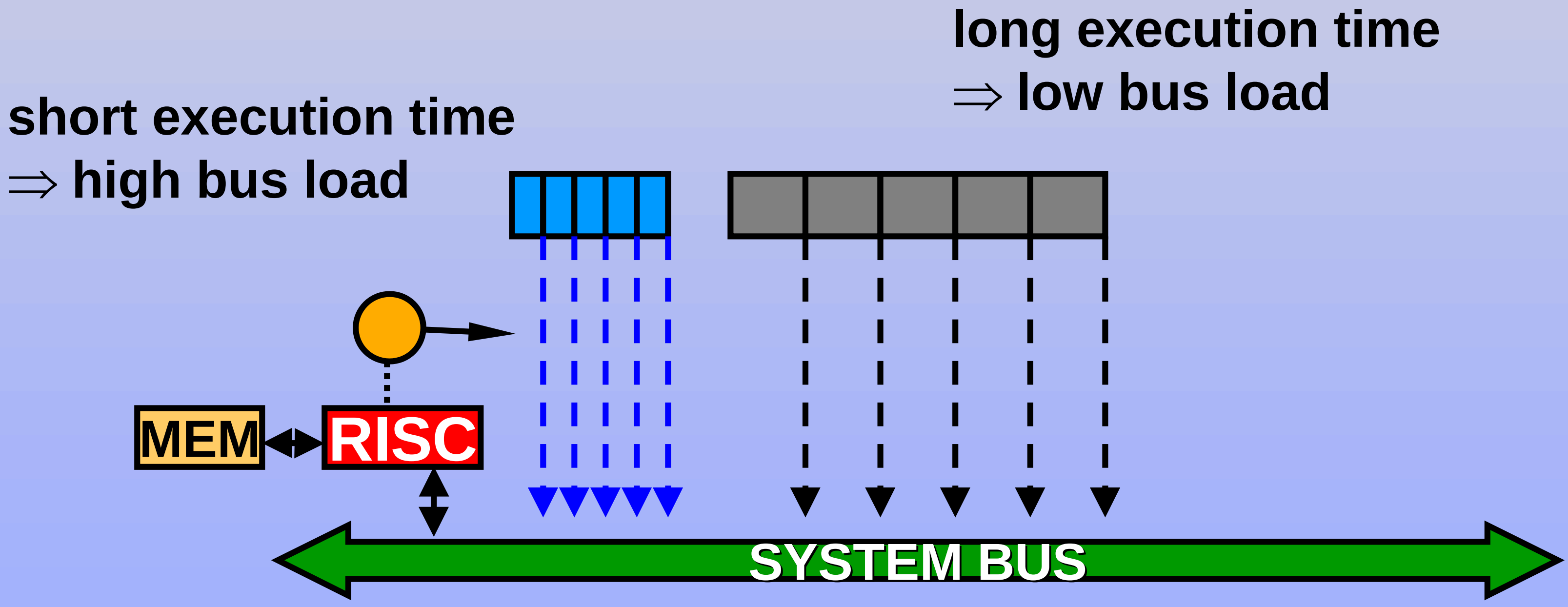
Complex run-time interdependencies



- run-time dependencies of independent components via communication
- influence on timing and power

Interdependency example

- complex heterogeneous systems
- complex non-functional interdependencies
- complex system corner cases



MPSoC platform verification - state of the art

- current approach: Target architecture co-simulation
  - combines functional and performance validation
  - reuse component validation pattern for system integration and function test
  - reuse application benchmarks for target architecture function validation
  - visualization of system execution
  - extensive simulation run times to include many test cases

Co-simulation limitations

- identification of *system* performance corner cases
  - different from *component* performance corner cases
  - target architecture behavior unknown to the application function developer (cp. functional HW test)

    $\Rightarrow$ test case definition and selection ?
- analysis of target architecture
  - confusing variety of run-time interdependencies
  - data dependent “transient” run-time effects
  - mixed in co-simulation

    $\Rightarrow$ limited support of design space exploration

    $\Rightarrow$ debugging challenge
- inclusion of incomplete application specifications

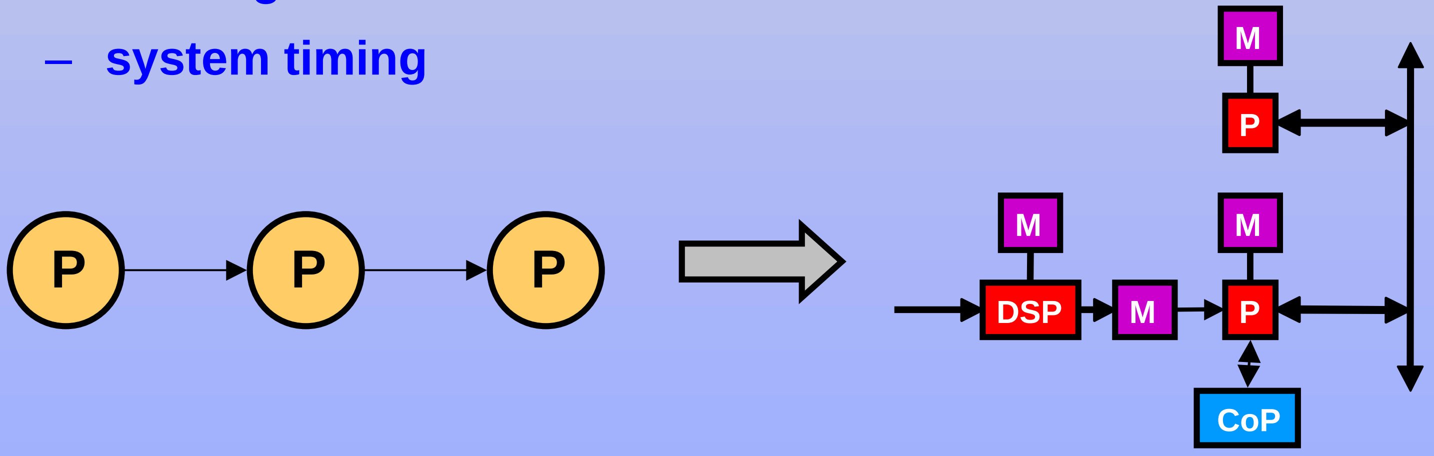
  $\Rightarrow$ additional performance models required

Lecture objective

- better understanding of target architecture run-time effects
- propose approaches to improve and formalize analysis

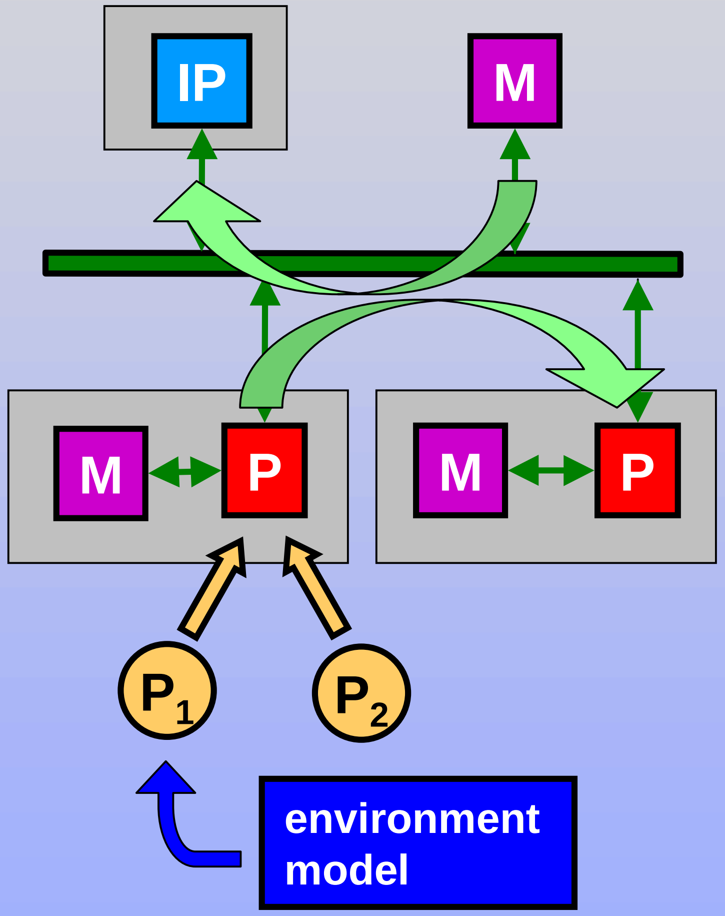
Target architecture analysis

- given
  - an application and its environment modeled by a set of communicating processes
  - a heterogeneous HW/SW target architecture
  - an implementation of the processes on the architecture
- model and analyze
  - the target architecture information flow
  - system timing



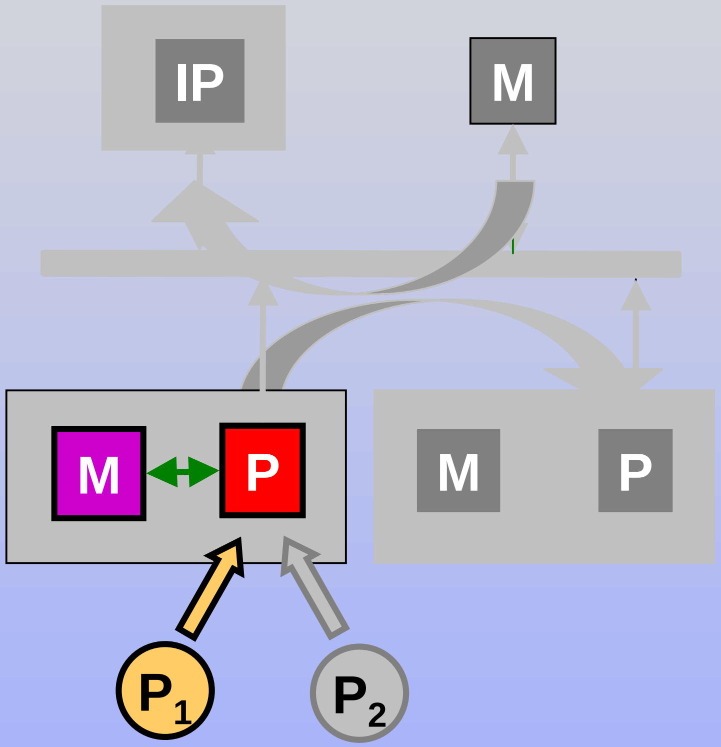
Timing parameters

- architecture component timing
- subsystem timing
- system timing



2 Architecture component modeling&analysis

- architecture component timing
  - process execution timing
  - communication timing



- processing elements
- memories
- interconnect

Processing elements

- **processing elements**
  - **fully programmable components (processors)**

    DSP **RISC** **µC** **VLIW**

  - **weakly programmable coprocessors**
    components with selectable, predefined control sequences, possibly with chaining (FP coprocessor, graphics processor, DMA, ...)

    multi-channel module **image coprocessor**

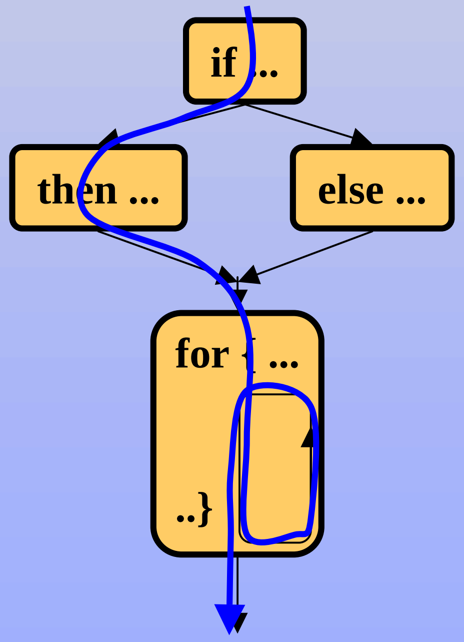
  - **hard coded function components**

    CAN bus interface **ADC** **VLD coprocessor**

Processing element timing

- processing element timing and communication determined by

- execution path
  - control data dependent
  - input data dependent



- function implementation
  - component architecture
  - software architecture
  - compiler or synthesis

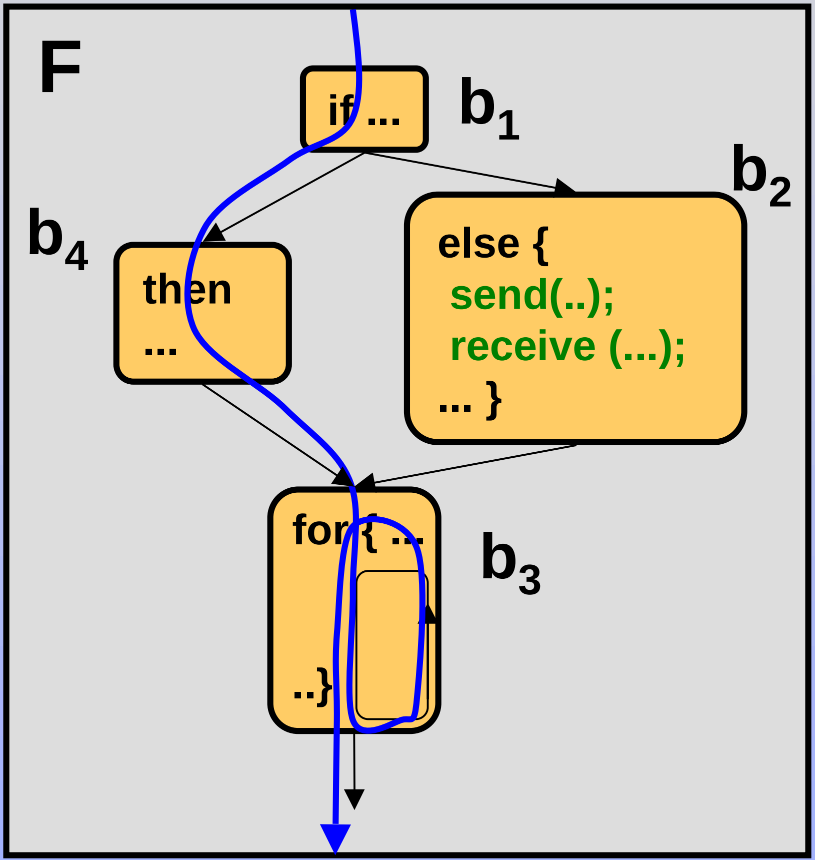
Processing element timing - 2

- process timing can be evaluated by
  - simulation/performance monitoring, e.g. using break points
    - stimuli, e.g. from component design
    - data dependent execution $\rightarrow$ upper and lower timing bounds
  - simulation challenges
    - coverage?
    - cache and context switch overhead due to run-time scheduling with process preemptions
    - influence of run-time scheduling depending on external event timing
  - formal analysis of individual process timing
    - serious progress in recent years

$\Leftrightarrow$ process timing can be approximated or just estimated (cp. VCC processor models)

Formal execution path timing analysis

- **execution path timing**



$$t_{pe}(F, pe_j) = \sum_I t_{pe}(b_i, pe_j) \cdot c(b_i)$$

b_i **basic block** (Li/Malik)
or program segment (Ye/Ernst)

$t_{pe}(b_i,pe_j)$ execution time of b_i
on processing element pe_j

$c(b_i)$ execution frequency of b_i
path dependent

worst/best case timing bounds solved e.g. as ILP problem with flow analysis (e.g. Li/Malik, Wolf/Ye/Ernst, ...)

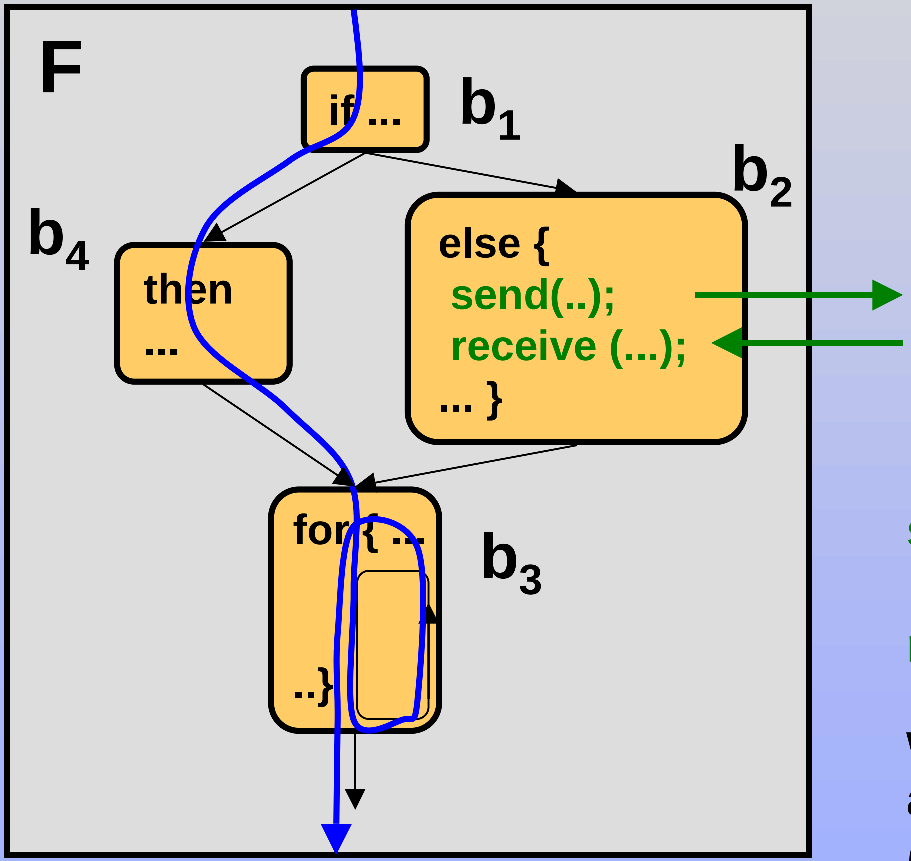
Implementation influence in $t_{pe}(b_i)$

- $t_{pe}(b_i, pe_j)$ determined by
  - processing element architecture
  - compiler / HW synthesis
  - API software
- $t_{pe}(b_i, pe_j)$ analysis can use
  - instruction execution table
  - abstract execution model
  - local b_i simulation

$\left.\begin{array}{l}\text{instruction execution table}\\\text{abstract execution model}\end{array}\right\}$ requires compiler code analysis

Process communication

- **execution path communication**



$$s(F) = \sum_{i} s(b_i) \cdot c(b_i)$$

$$r(F) = \sum_{i} r(b_i) \cdot c(b_i)$$

$s(b_i)$ **sent data in b_i**

$r(b_i)$ **received data in b_i**

worst/best case communication analysis solved e.g. as ILP problem (Wolf/Ernst)

Implementation influence in $r(b_i)$ and $s(b_i)$

- $r(b_i)$ and $s(b_i)$ determined by
  - data volume
  - data encoding
  - communication protocol

Interconnect timing

- interconnect timing can be evaluated by
  - simulation, cp. process element timing
  - statistical load data
  - simple formal models, e.g. for TDMA (e.g. MicroNetwork (Sonics))

⇔ interconnect timing can be approximated or just estimated

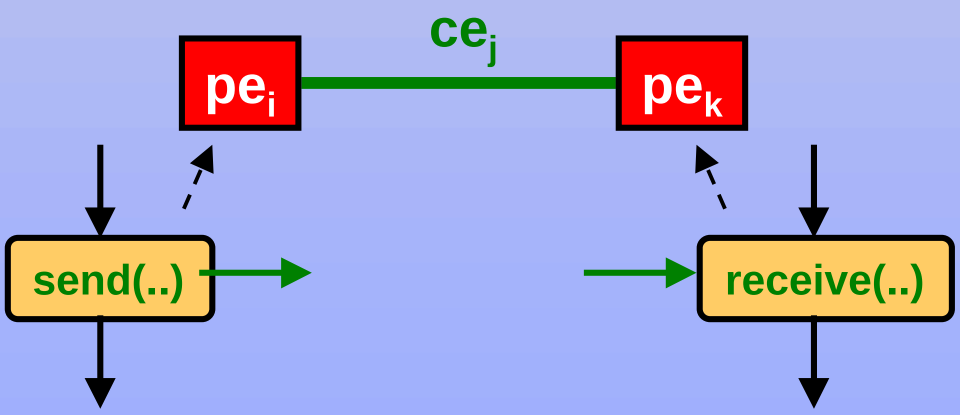
Formal interconnect analysis

- **word transfer**

$$t_{com}(s(x),ce_j) = s(x) \cdot t_{com}(s(1),ce_j)$$

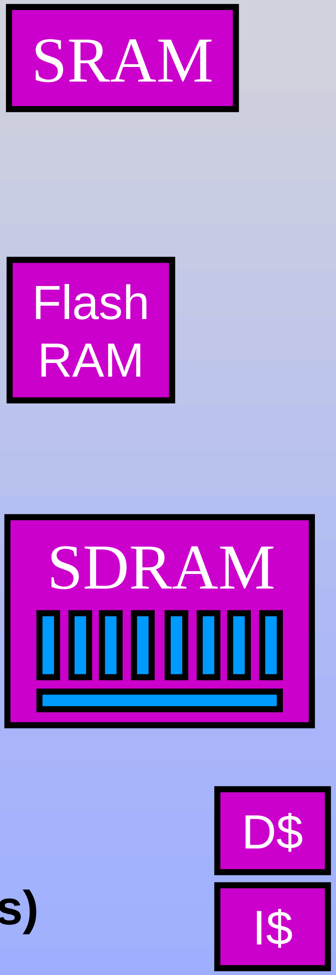
- **packet transfer (simplified: fixed length pl)**

$$t_{com}(s(x),ce_j) = \left\lceil \frac{s(x)}{pl_{ce_j}} \right\rceil \cdot t_{com}(s(pl_{ce_j}),ce_j)$$



Memories

- SRAM
  - equal access time
  - no control overhead
- FLASH, ROM, EPROM, EEPROM
  - asymmetric read and write
  - similar to SRAM otherwise
- DDRAM, RDRAM, SDRAM, SRAM, ...
  - multiple banks
  - burst access (packets)
- Cache
  - various control mechanisms
  - burst access to background (cache lines)



Memory models

- SRAM timing here included in t_{pe}
  - program memory access as instruction fetch time
  - data memory access as $t_{load/store}$
- SDRAM with cache
  - t_{hit} included in t_{pe}
  - t_{cmiss} (cm_i): miss time of cache memory cm_i

$$t_{cmiss}(cm_i) = t_{com}(cm_i, ce_k) + t_m(m_j)$$

t_{com} (cm_i, ce_k): communication time for cache line transfer over ce_K

t_m (mj) : memory access time for m_j

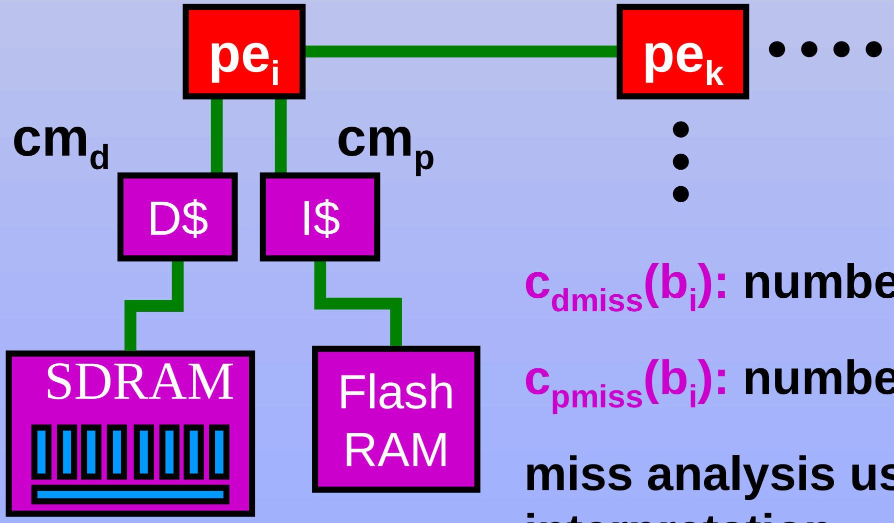
Cache miss overhead

- cache miss overhead
  - data cache cm_d

$$t_{dcache}(F, cm_d) = t_{cmiss}(cm_d) \cdot \sum_{b_i \in F} c_{dmiss}(b_i)$$

  - program cache cm_p

$$t_{pcache}(F, cm_p) = t_{cmiss}(cm_p) \cdot \sum_{b_i \in F} c_{pmiss}(b_i)$$



$c_{dmiss}(b_i)$: number data cache misses in b_i

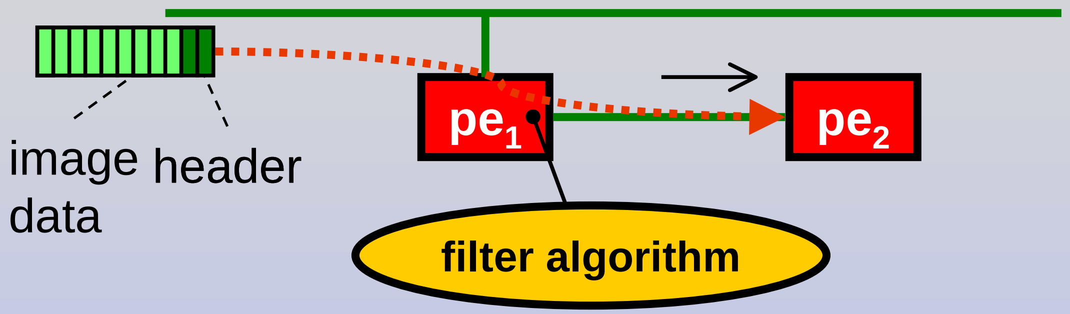
$c_{pmiss}(b_i)$: number program cache misses in b_i

miss analysis uses data flow analysis, abstract interpretation, ...

Improved process timing model

- state and „context“ consideration
- modeled as process “mode”
- example: image filter

Image filter example



- **image filter process**
  - **receives packet with header and image data**
  - **performs address match verification**
  - **filters picture data**
  - **forwards filtered picture data to pe$_2$**
- **execution contexts (picture size): not considered, large picture, small picture**
- **execution contexts (address): not considered, address miss, address match**

Image filter results

Receive Data [kB] Send Data [kB]		Address not considered	Address miss	Address match
Size not considered	Rec	[6.2 , 25.0]	[6.2 , 25.0]	[6.2 , 25.0]
	Snd	[0 , 24.4]	[0 , 0]	[5.9 , 24.4]
Large Picture	Rec	[25.0 , 25.0]	[25.0 , 25.0]	[25.0 , 25.0]
	Snd	[0 , 24.4]	[0 , 0]	[24.4 , 24.4]
Small Picture	Rec	[6.2 , 6.2]	[6.2 , 6.2]	[6.2 , 6.2]
	Snd	[0 , 5.9]	[0 , 0]	[5.9 , 5.9]

Tight send and receive data rate intervals [min, max] of the filter process (results: SYMTA)

Image filter intervals

Timing [ms]	Address not considered	Address miss	Address match
Size not considered	[5 , 681]	[6 , 40]	[38 , 681]
Large Picture	[19 , 572]	[20 , 39]	[265 , 572]
Small Picture	[5 , 67]	[6 , 13]	[38 , 64]

Timing intervals [min, max]
for StrongARM architecture incl. caches (results: SYMTA)

Timing and communication model

- What timing and communication model is appropriate?
  - – worst case?
  - – min/max (interval)?
  - – typical?
  - – statistics?
  - $\Rightarrow$ more information needed

Subsystem & component summary

- analysis of individual process timing as a first step
  - used as a basis for activation and resource sharing model
  - approach borrowed from RTOS
- include local memories and local communication

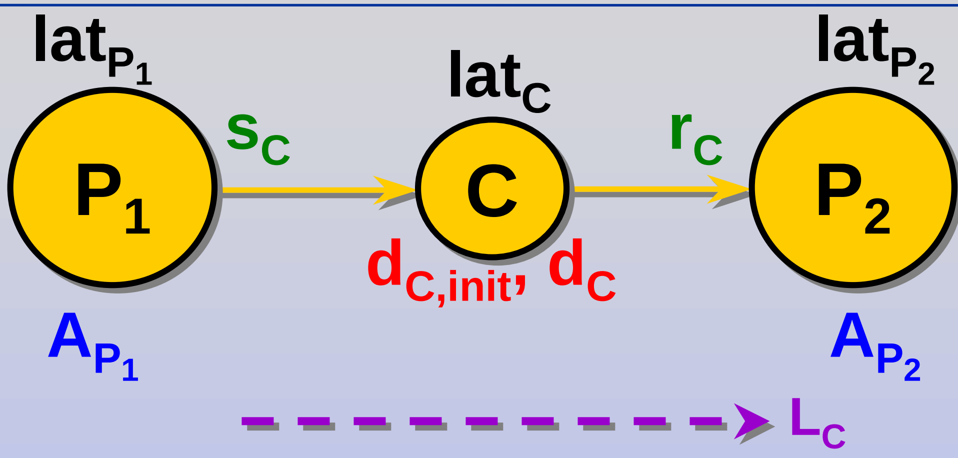
3 Process execution modeling

- component data required for execution modeling
  - process execution time t_{pe} (F, pe_j)
  - communication load r, s
  - communication timing t_{com} (s, ce_k)
  - process activation function
  - example: SPI - System Property Intervals
- environment model

Component parameter model

- example: SPI - System Property Intervals
  - coordination language for process system modeling
    - abstracts from detailed functionality
    - captures essential properties for component data timing, communication, and activation
    - properties represented as intervals
    - coordination uses process modes and virtual elements
    - originally used to combine models of computation

SPI - Parameters



- latency time intervals lat_{P_1}, lat_{P_2}, lat_C
- data rate intervals s_C, r_C
- activation functions A_{P_1}, A_{P_2}

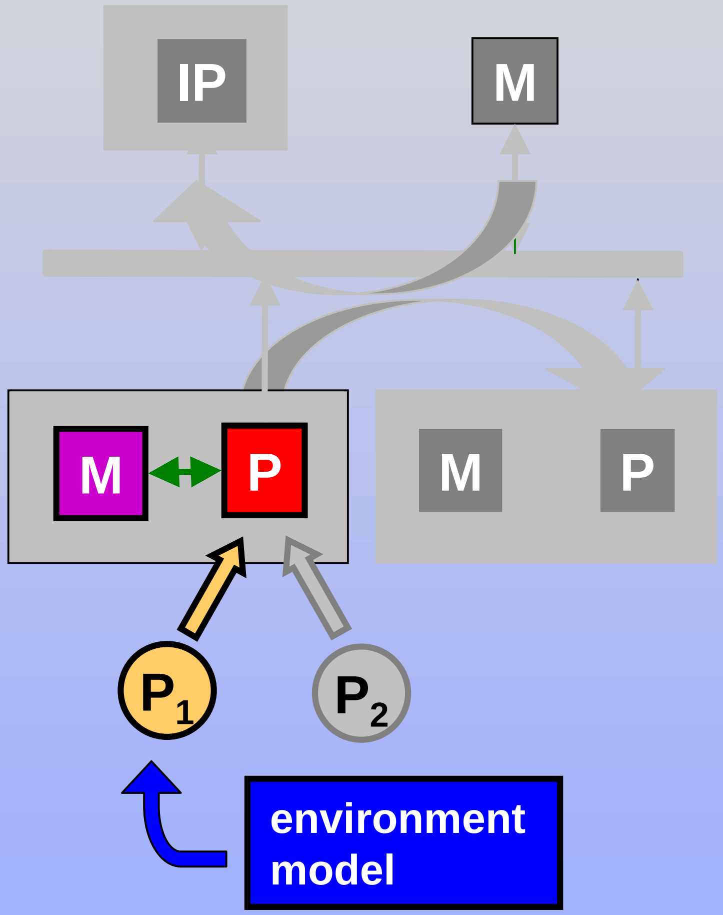
} process mode dependent

- data intervals on communication channels $d_{C,init}$, d_C
- process execution modes consider execution context

www.SPI-project.org

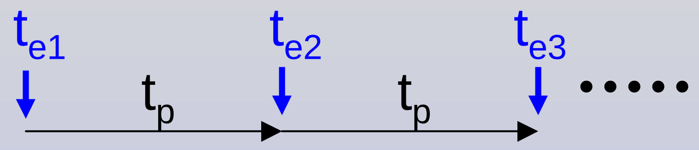
Timing parameters - environment

- architecture component timing
  - process execution timing
  - communication timing
  - activation -> environment model



Environment model

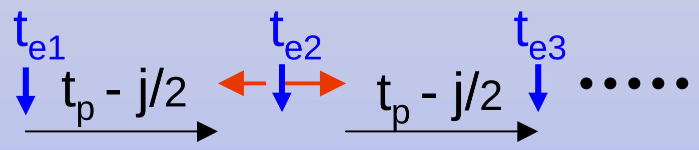
- **periodic events**



$$t_{e_i+1} - t_{e_i} = t_p$$

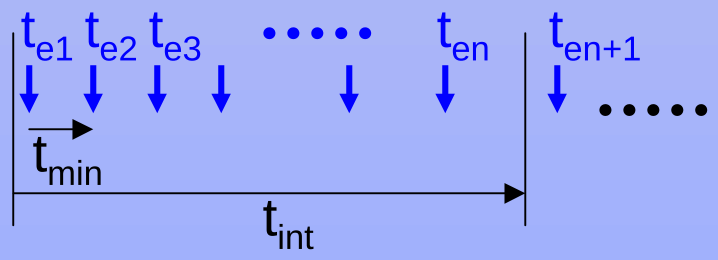
t_{ei} typically timer released

- **periodic events with jitter**



$$t_p - \frac{j}{2} \le t_{e_{i+1}} - t_{e_i} \le t_p + \frac{j}{2}$$

- **events with minimum inter arrival times**
  - **burst events, packets, sporadic events, etc.**



$$t_{e_{i+n+1}} - t_{e_i} \ge t_{\text{int}}$$

$$t_{e_i+1} - t_{e_i} \ge t_{\min}$$

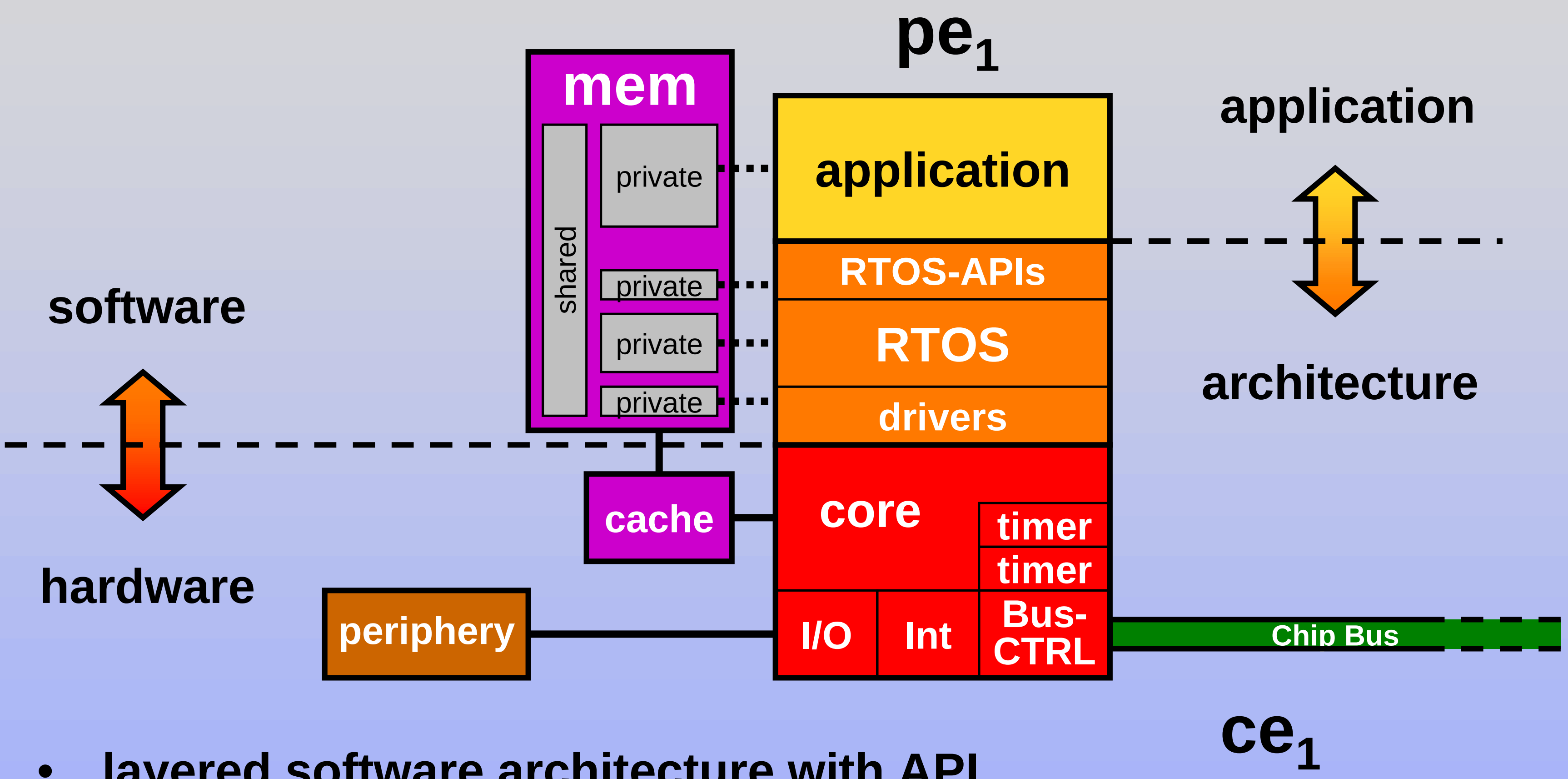
Environment model

- abstract event stream models instead of individual events
- classification of event patterns

Execution modeling summary

- event model defines the frequency and context of process activation
- event frequency given as interval, worst case or statistical value

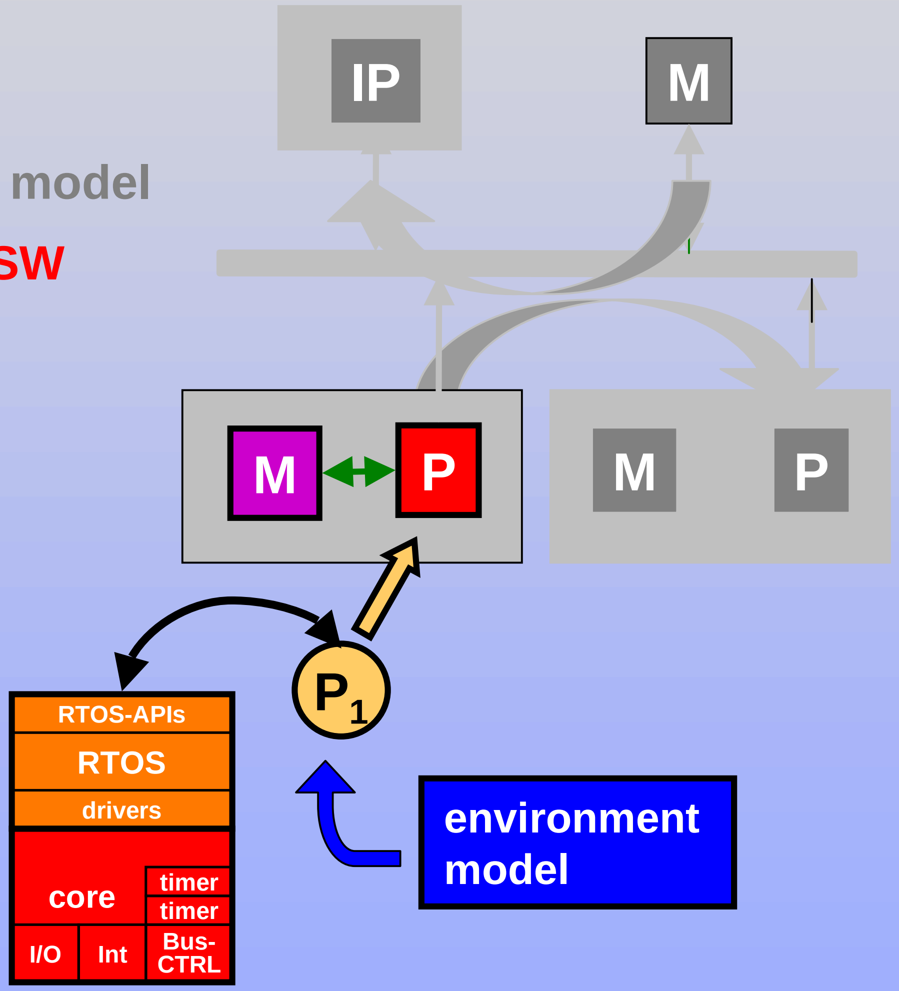
4 Software architecture



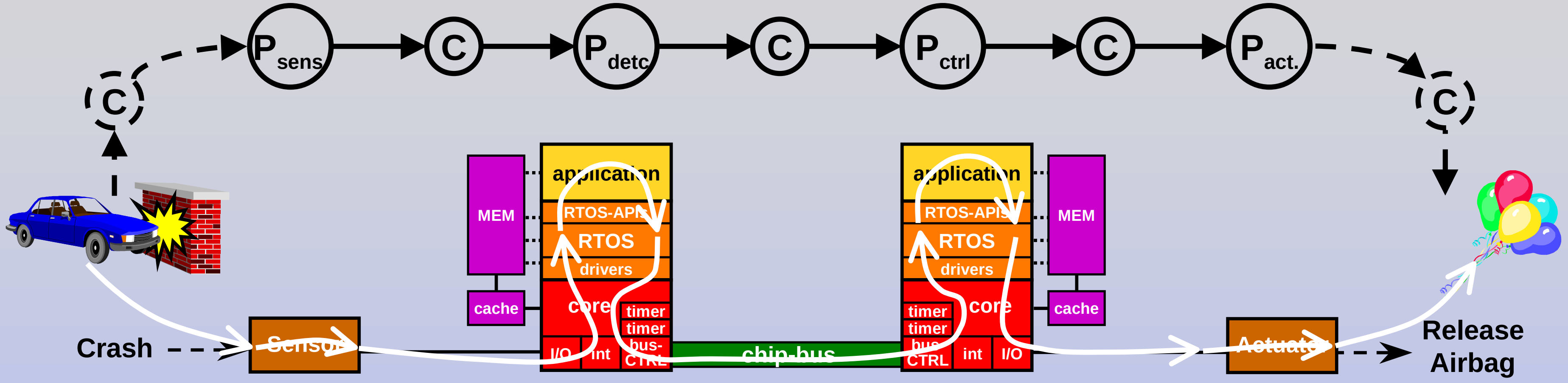
- layered software architecture with API
- consider function call hierarchy

Timing parameters - SW architecture

- architecture component timing
  - process execution timing
  - communication timing
  - activation -> environment model
  - timing includes all HW & SW components for process execution



Application example



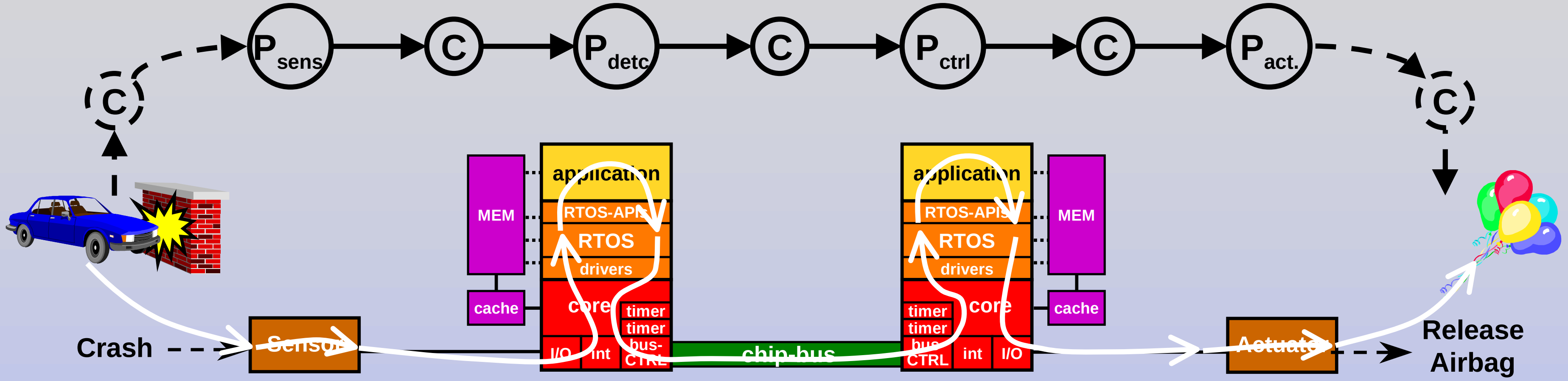
reaction time of airbag after crash ?

$$\underbrace{t_{crash} + t_{sens}}_{\text{physical delay}} + t_{csens} + t_{detc} + t_{fbus} + t_{ctrl} + t_{cact} + \underbrace{t_{act} + t_{airbag}}_{\text{physical delay}}$$

include SW architecture in process model

- **resolve APIs, drivers, OS calls, memory accesses, etc.**
- **include multi-hop communication**

Timing refinement



Reaction time of airbag after crash ?

$$\underbrace{t_{crash} + t_{sens}}_{\text{physical delay}} + t_{csens} + t_{detc} + t_{fbus} + t_{ctrl} + t_{cact} + \underbrace{t_{act} + t_{airbag}}_{\text{physical delay}}$$

$t_{csens} = t_{com} + t_{drv}$

$t_{detc} = t_{API} + t_{process} + t_{API}$

$t_{fbus} = t_{drv} + t_{com} + t_{drv}$

$t_{ctrl} = t_{API} + t_{process} + t_{API}$

$t_{cact} = t_{drv} + t_{com}$

t_{exe} = f (code, core, cache, mem, etc.)

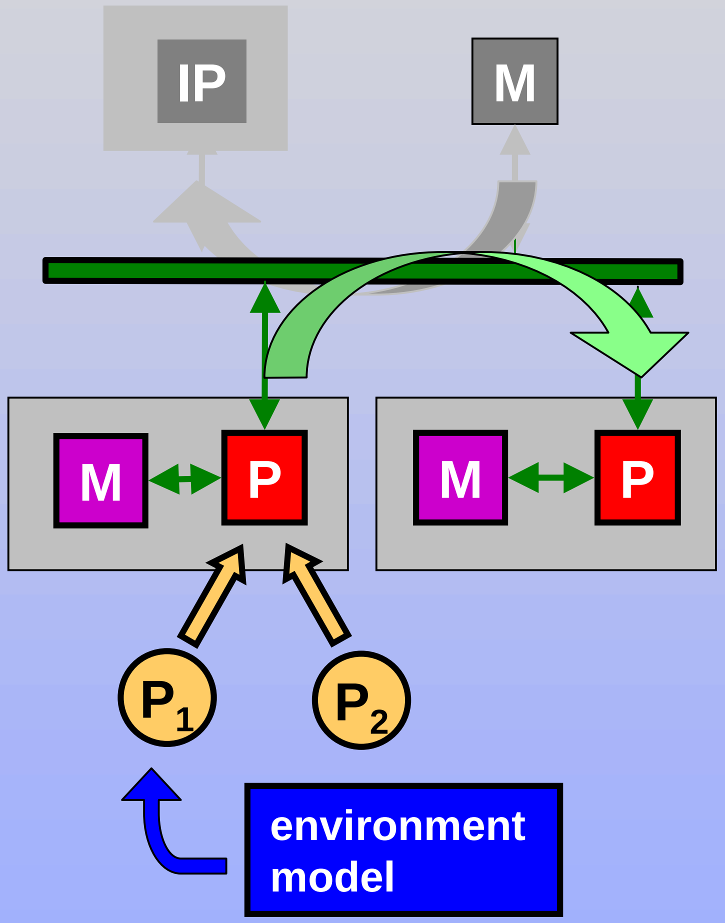
t_{com} = f (amount of data, width, speed, etc.)

5 Modeling shared resources

- resource sharing requires
  - resource arbitration - scheduling
    - static or dynamic order of processes
    - preemptive (interrupt) or non-preemptive („run to completion“)
  - context switching
    - on pe: process context switch
    - on ce: packet or connection setup overhead
    - in memory: similar to ce
  - context switching time t_{csw} can be determined at design time
  - resource arbitration effects are run-time dependent

Timing parameters - resource sharing

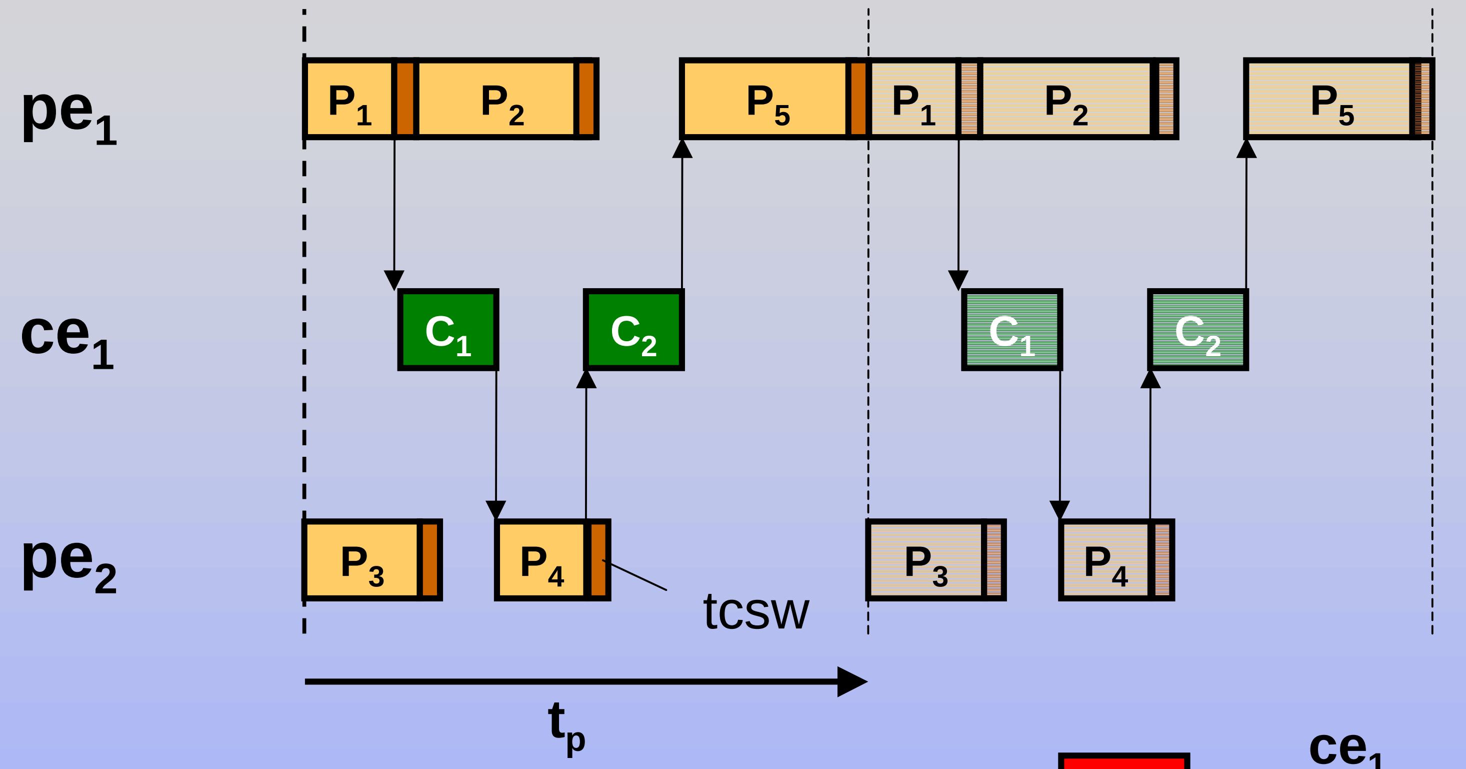
- architecture component timing
- subsystem timing
  - resource sharing
    - process scheduling
    - communication scheduling



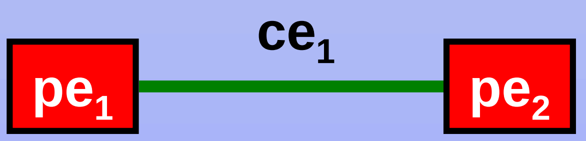
Scheduling strategies

- static execution order
- time driven scheduling
  - fixed
  - dynamic
- priority driven scheduling
  - static priority assignment
  - dynamic priority assignment
- efficiency depends on environment model (periodic, jitter, burst)

Static execution order scheduling



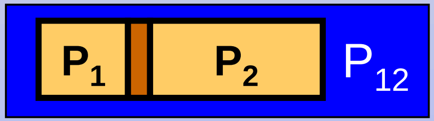
t_p: scheduling period



architecture example

Static execution order scheduling - 2

- execution time: sum of process times + t_{CSW} + t_{com}
- best suited if timing and control are input data independent
- supports
  - interleaved resource utilization
  - buffer size optimization
  - compiler optimization across processes
  - process context exploitation
- application example:
  - DSP



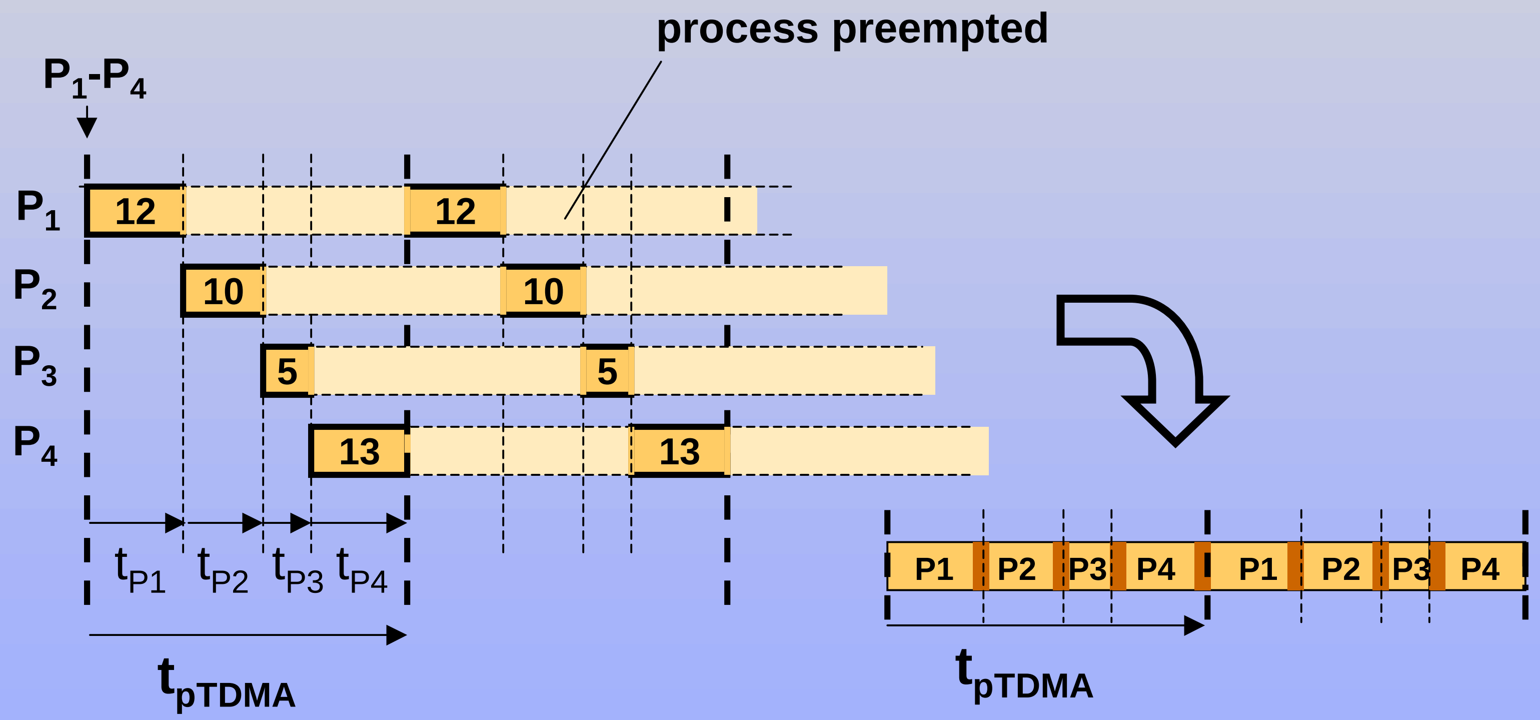
merged processes

Static execution order scheduling - 3

- different event timing models
  - periodic input events $\rightarrow$ periodic output events with jitter
  - sporadic events $\rightarrow$ sporadic output events
- static order problems
  - dynamic environments: jitter, bursts
  - deadline requirements
  - processes/communication with context dependent timing

Time driven scheduling

- time division multiple access (TDMA)
  - periodic assignment of fixed time slots
  - applicable to pe or ce



TDMA example

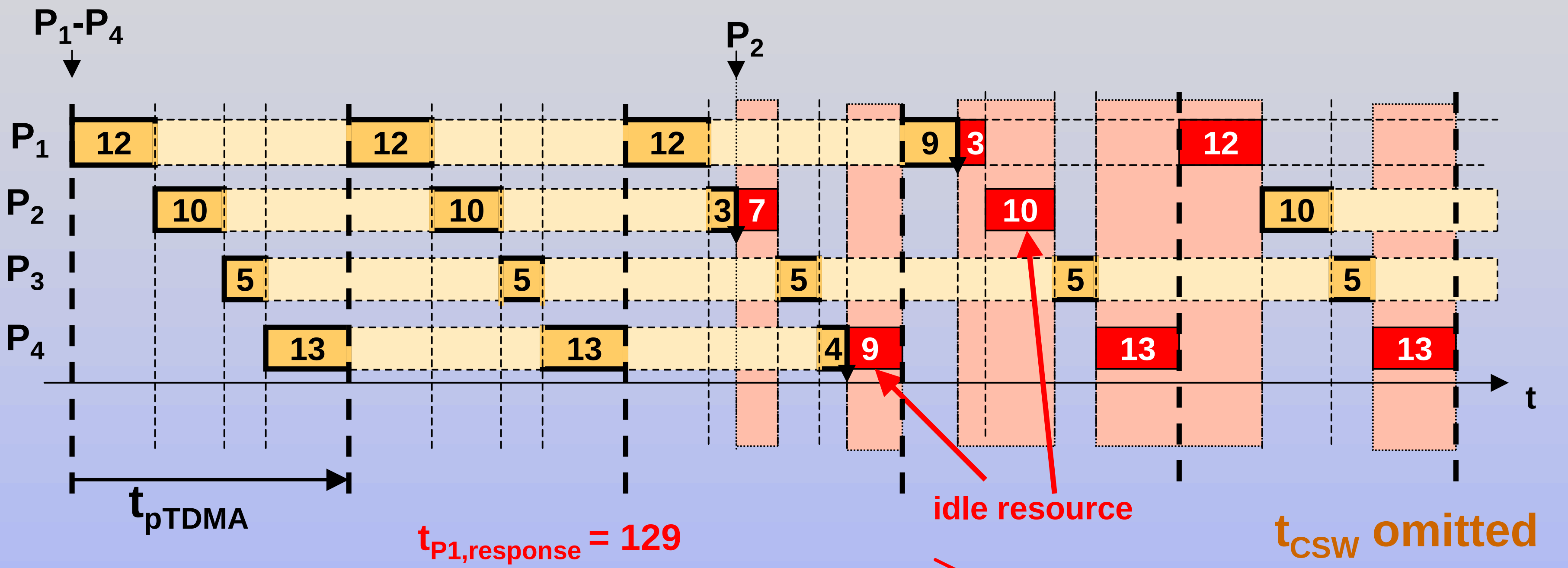
TDMA

- predictable and independent performance down scaling allows to merge individual solutions

$$t_{peTDMA}(P_i, pe_i) = \left\lfloor \frac{t_{pe}(P_i, pe_i) - t_{csw}}{t_{Pi}} \right\rfloor \cdot t_{pTDMA} + t_{pe}(P_i, pe_i) \bmod t_{pi}$$

- time slot size adaptable to different service levels
- supports all input event timing models
- generates output jitter as a result of execution times
- problems
  - utilization
  - extended deadlines

TDMA scheduling example

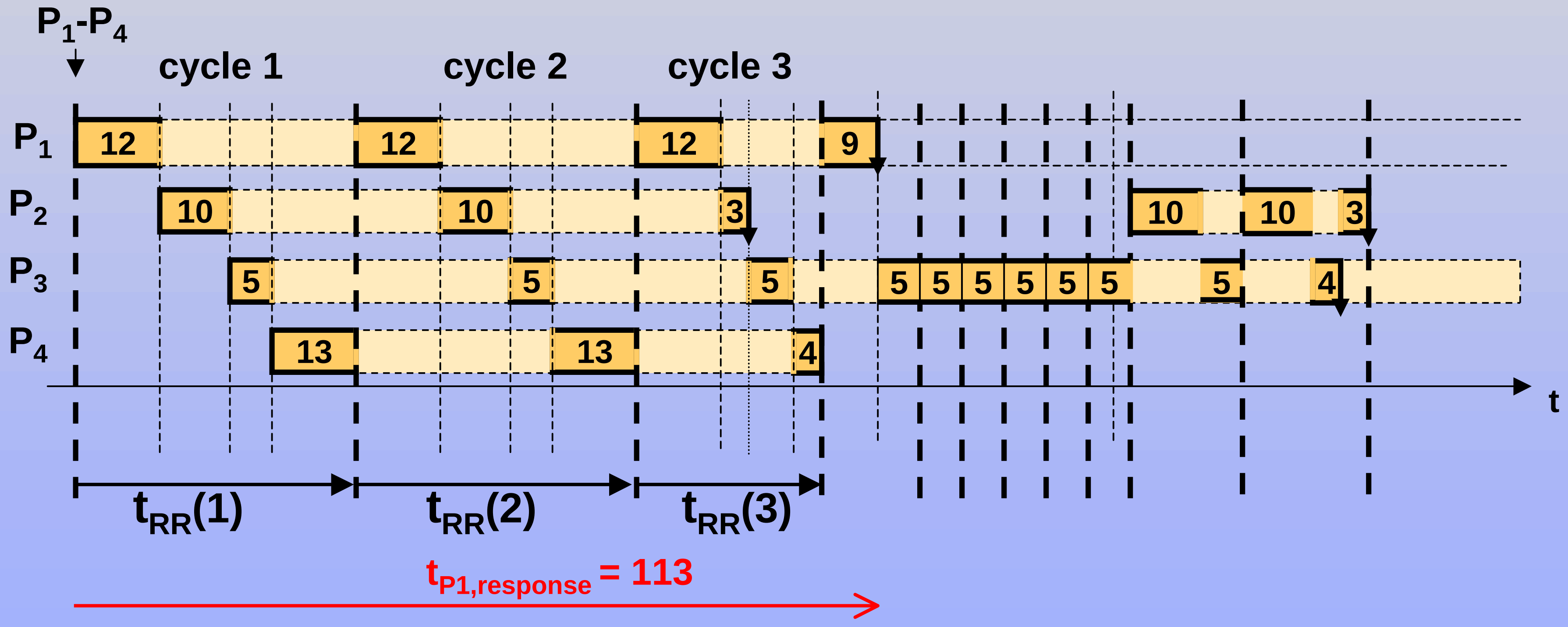


t_{CSW} omitted for simplicity

Scheduling and idle times in TDMA

Dynamic time driven scheduling

- example: Round Robin scheduling
  - cyclic process execution
  - resource released when task is finished or not activated



Round Robin example

Round Robin scheduling

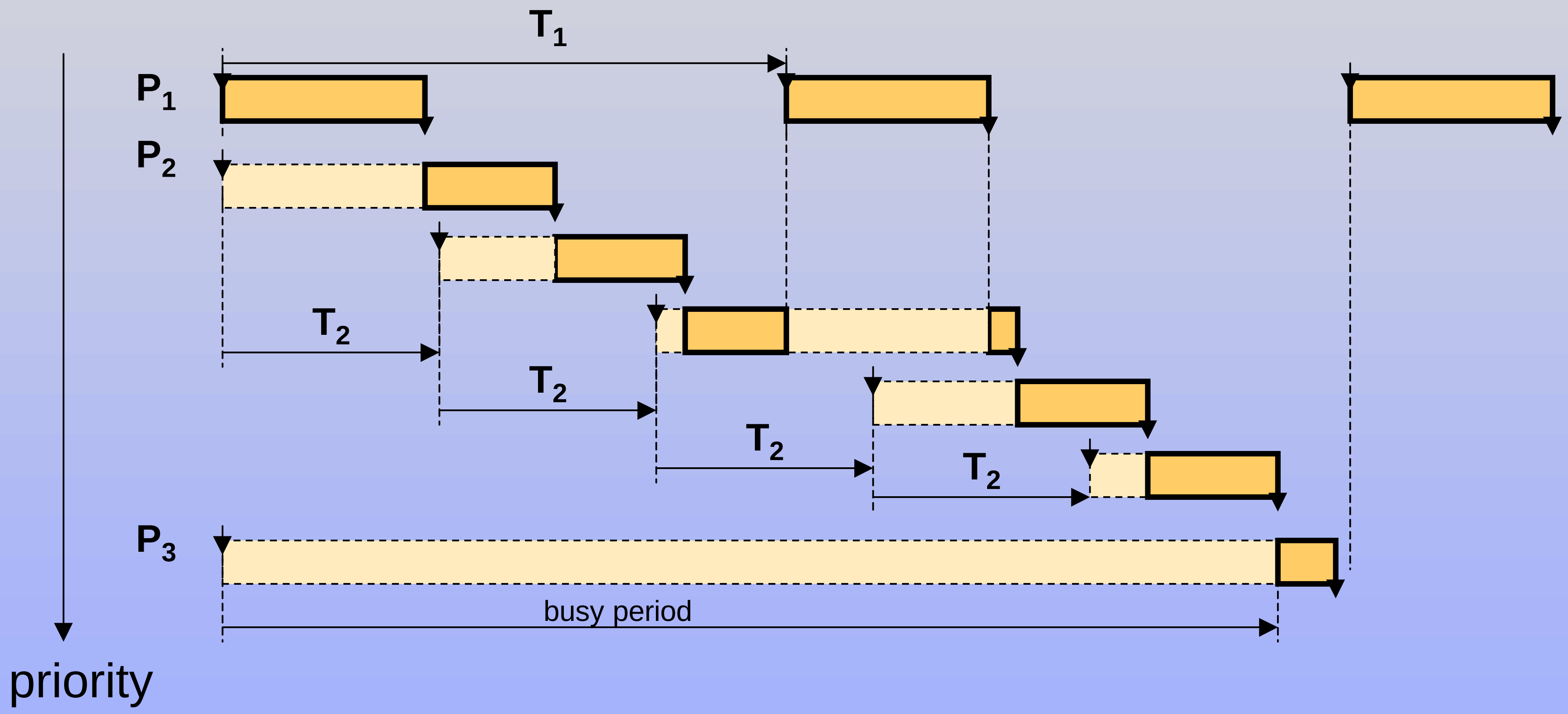
- no idle times - higher efficiency than TDMA
- guaranteed minimum resource assignment per process
  - appropriate e.g. for soft deadlines (“best effort”) and QoS requirements
- supports all input event timing models
- creates output jitter and possibly output bursts
- problems
  - process execution interdependency
  - timing analysis more difficult than TDMA

Priority driven scheduling

- Static priority assignment
  - model 1
    - multi rate periodic input events with jitter and deadlines at end of period
      - typical scheduling approach: Rate-monotonic scheduling (RMS, priority decreases with period)
      - optimal solution for single processors (Liu, Layland)
  - model 2
    - like model 1 but with process dependencies
      - solution for multiprocessors based on RMS: Yen, Wolf
  - model 3
    - like model 1 but arbitrary deadlines
      - timing analysis solution by Lehoczky
      - iterative algorithm for priority assignment (Audsley)
      - analysis for jitter and bursts (single processor) by Tindell

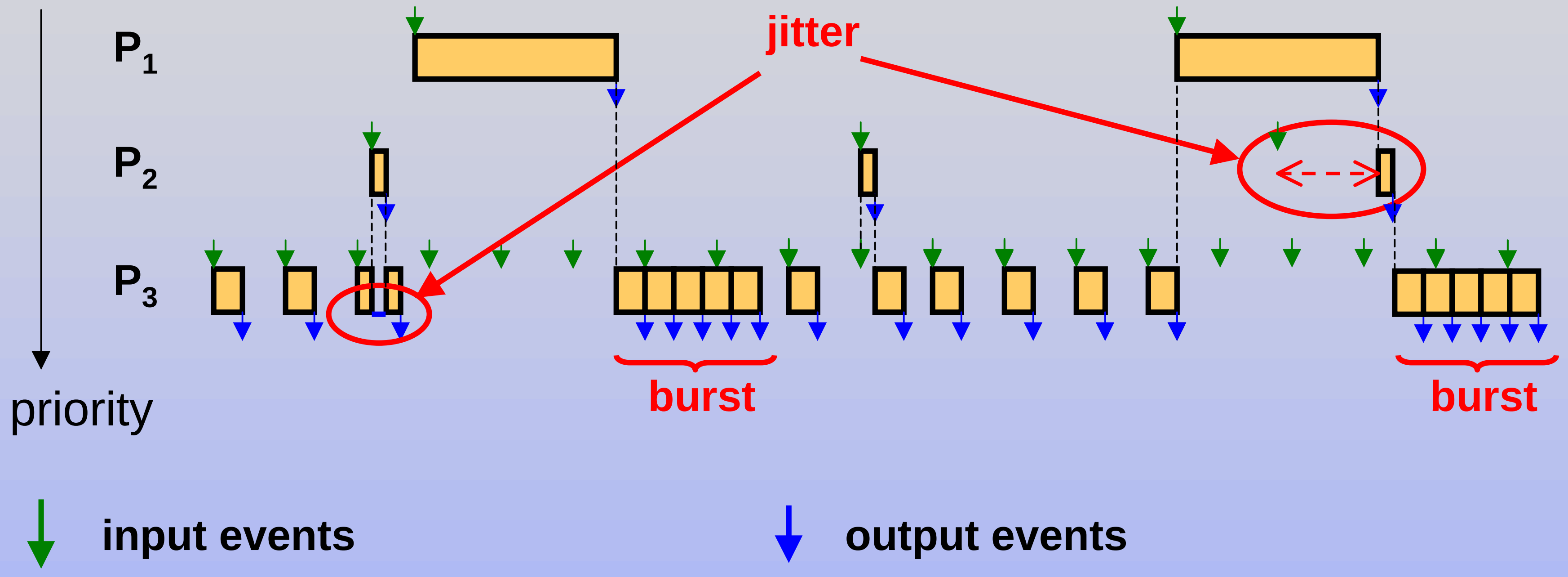
Model 3 - arbitrary deadlines

- scheduling with arbitrary deadlines - may create output bursts for periodic input events



Static priority scheduling with arbitrary deadlines

Burst generation



Output bursts in static priority scheduling with arbitrary deadlines and periodic input events

Dynamic priority assignment

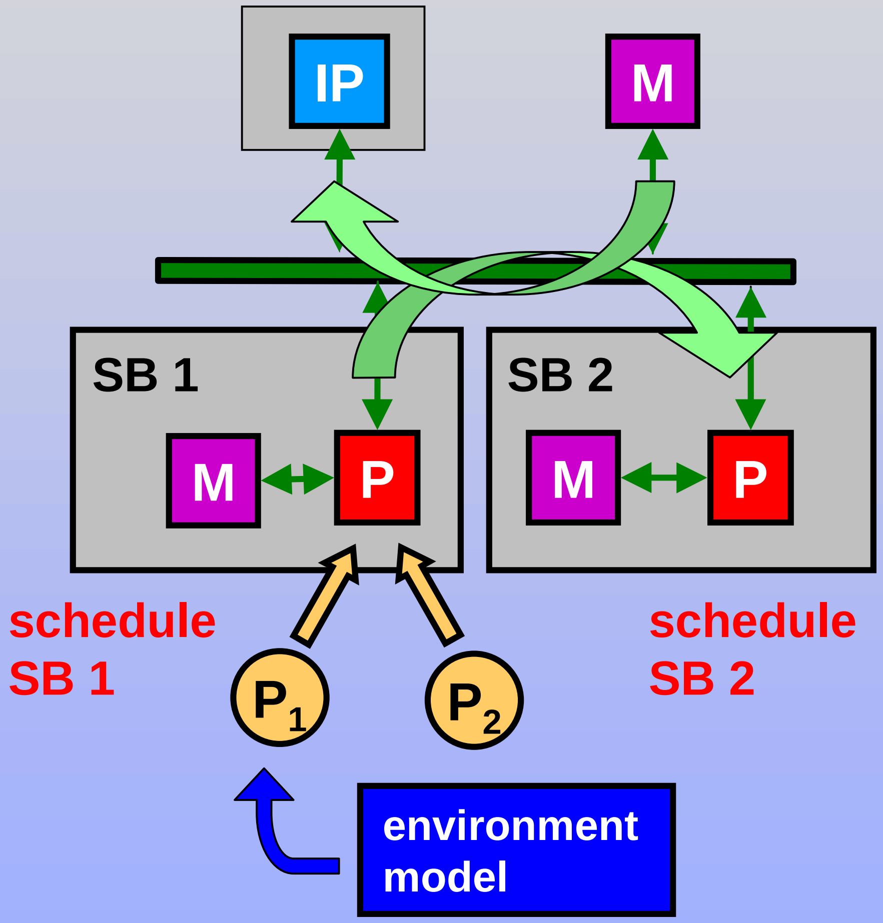
- priority assignment at run time
- optimal priority assignment: Earliest Deadline First (EDF)
- EDF adapts to input event timing
- problem:
  - requires run-time scheduler task $\rightarrow$ overhead
  - requires deadline determination (e.g. Ziegenbein, ICCAD 2000)
  - difficult to analyze - active research topic (e.g. load models)

Resource sharing - summary

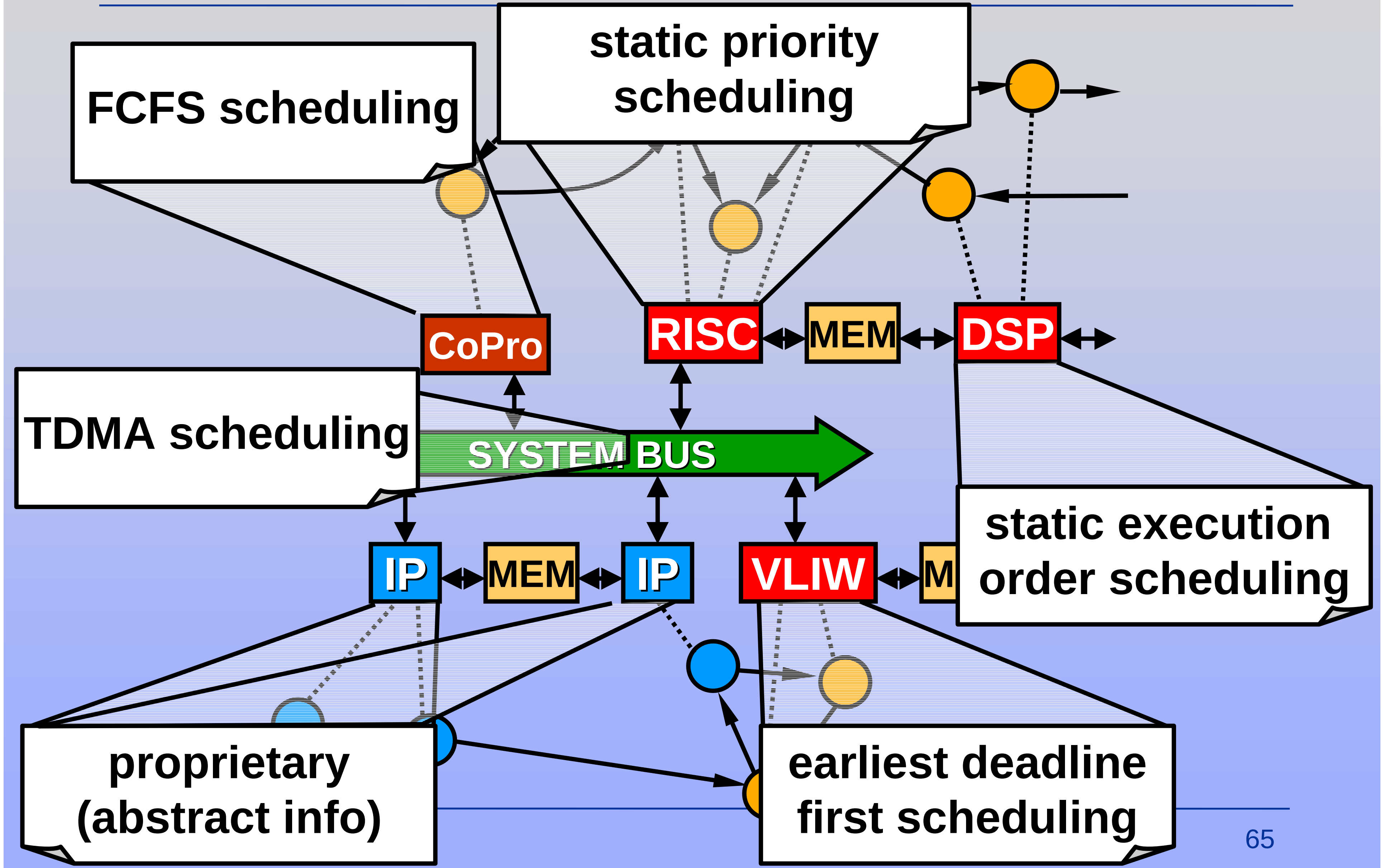
- individual component and process data required
- formal analysis for numerous scheduling strategies, event models and constraints available
  (even for manual calculation)
- results can be used to calculate
  - communication and processor load
  - timing, e.g. response times, output event models
  - memory requirements
  - power consumption

6 Architecture and global analysis

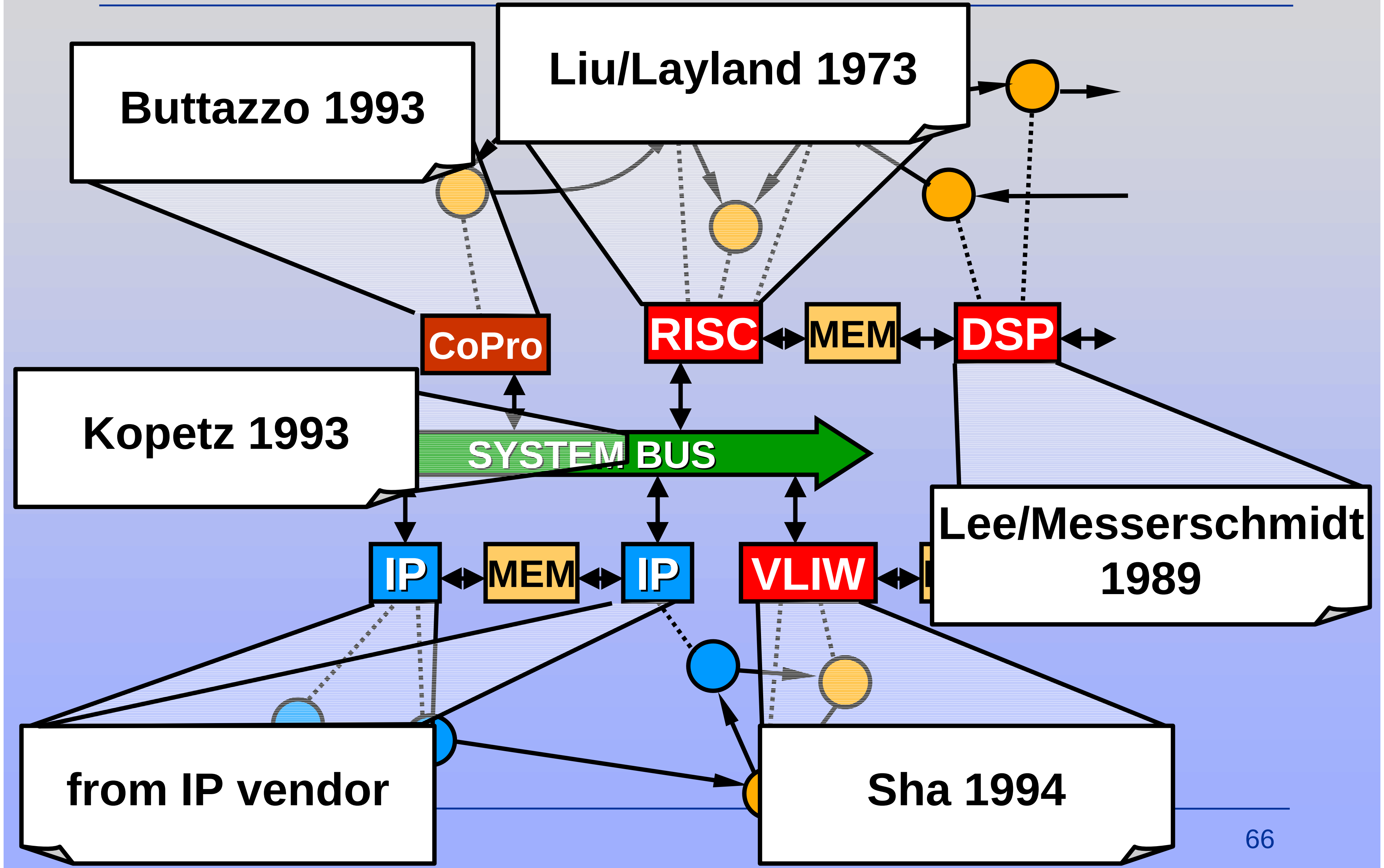
- architecture component timing
- subsystem timing
- system timing
  - timing of coupled subsystems



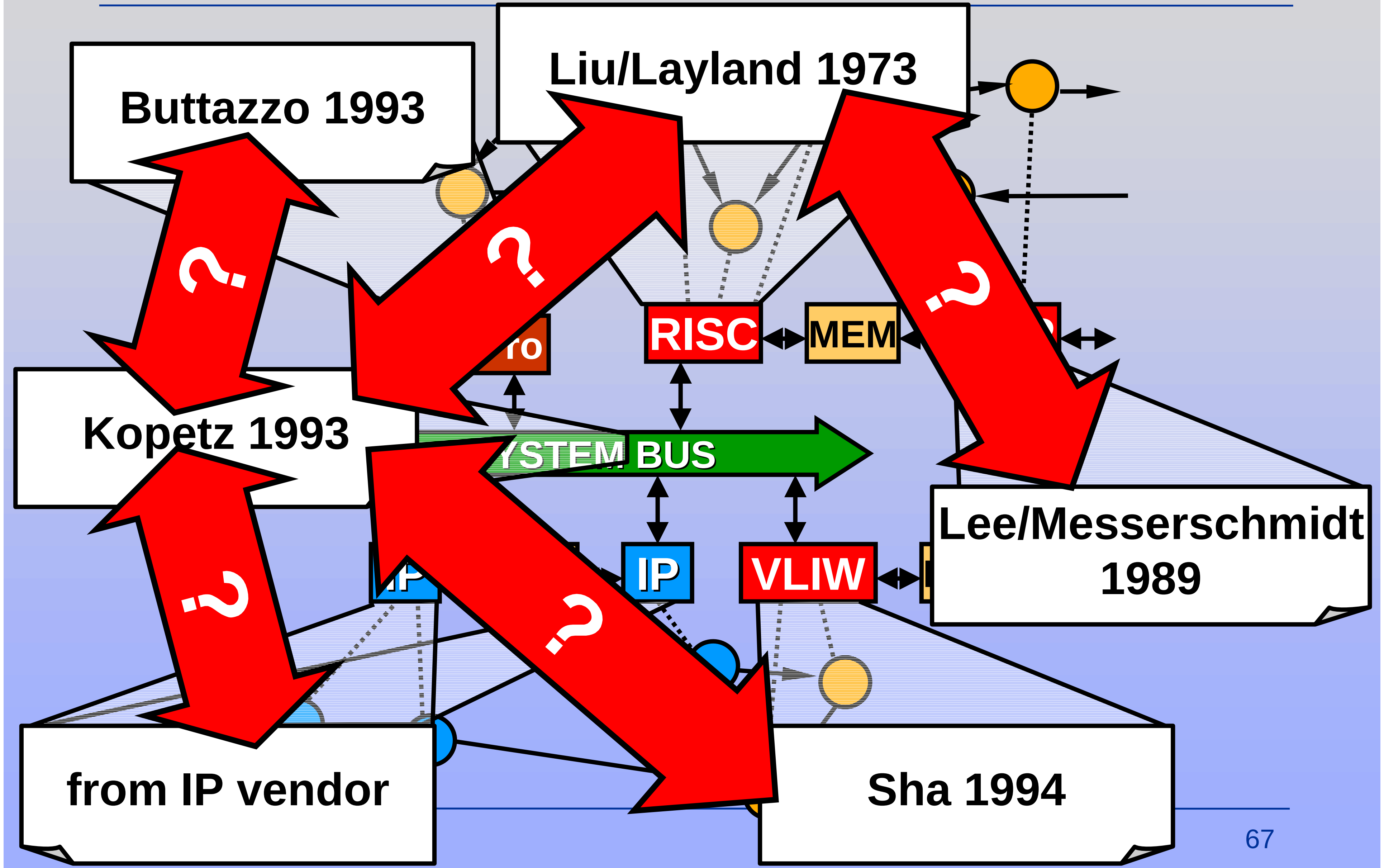
Multiple Scheduling Strategies


FCFS scheduling
static priority scheduling
CoPro
RISC
MEM
DSP
TDMA scheduling
SYSTEM BUS
static execution order scheduling
IP
MEM
IP
VLIW
M
proprietary (abstract info)
earliest deadline first scheduling

Corresponding Analysis Techniques

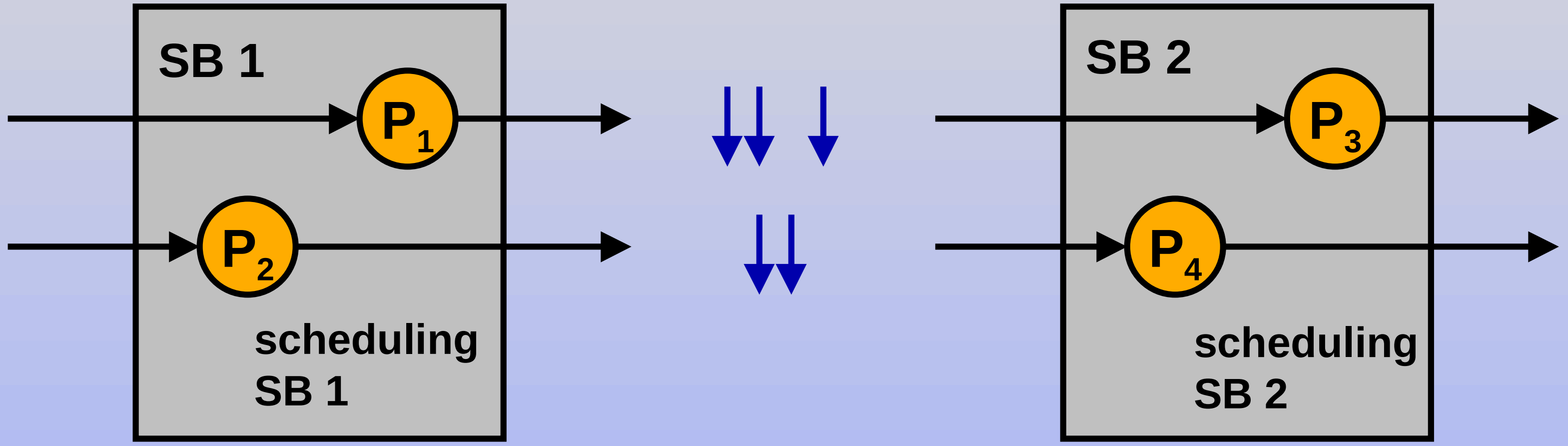


Integration ???



Subsystem coupling

- independently scheduled subsystems are coupled by data flow



⇒ subsystems coupled by event streams

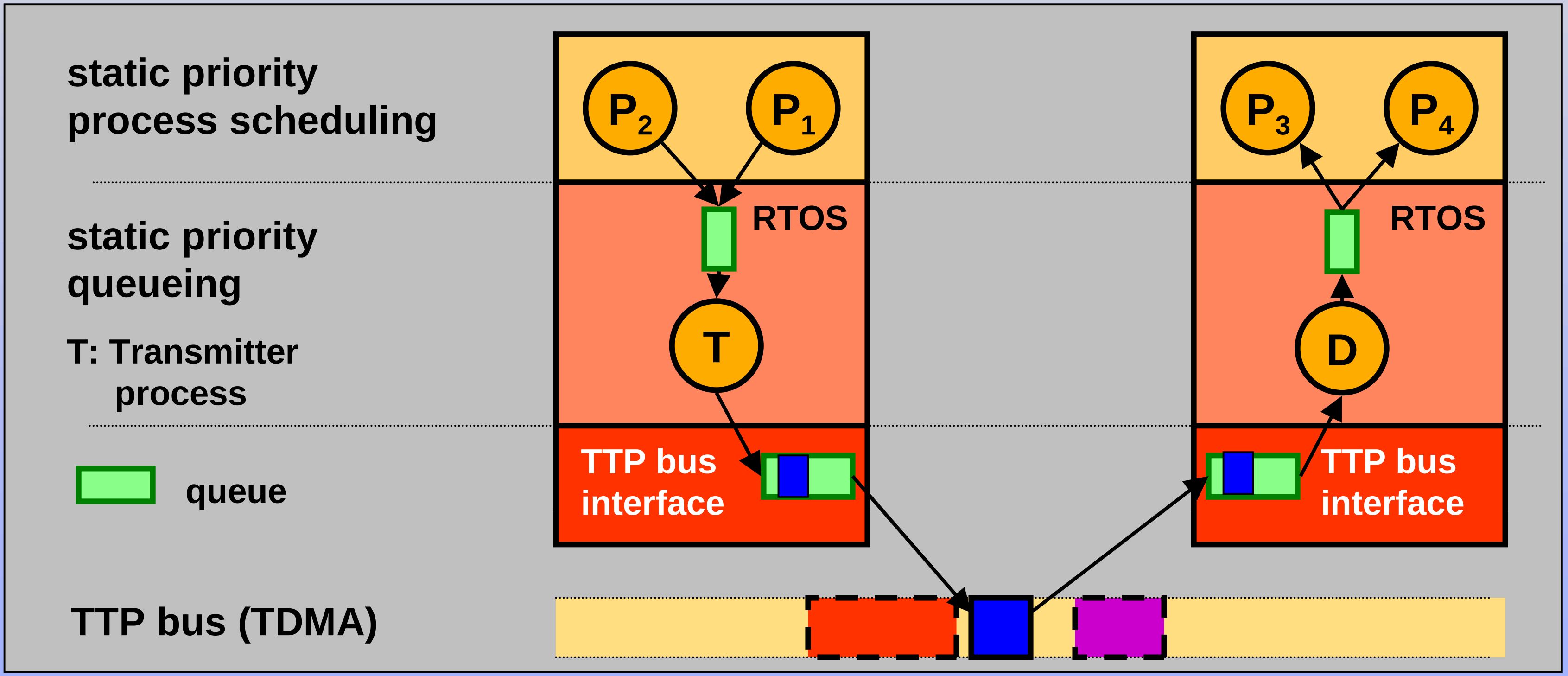
⇒ coupling corresponds to event propagation

System timing analysis approaches

- analysis scope extension to several subsystems
- event model generalization for a set of scheduling strategies
- event model adaptation

Analysis scope extension

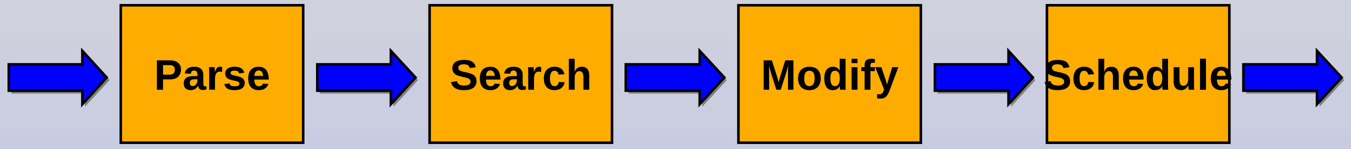
- coherent analysis („holistic“ approach)
- example: Tindell 94, Pop/Eles (DATE 2000): TDMA + static priority



- problem: large number of combinations possible!

Event model generalization

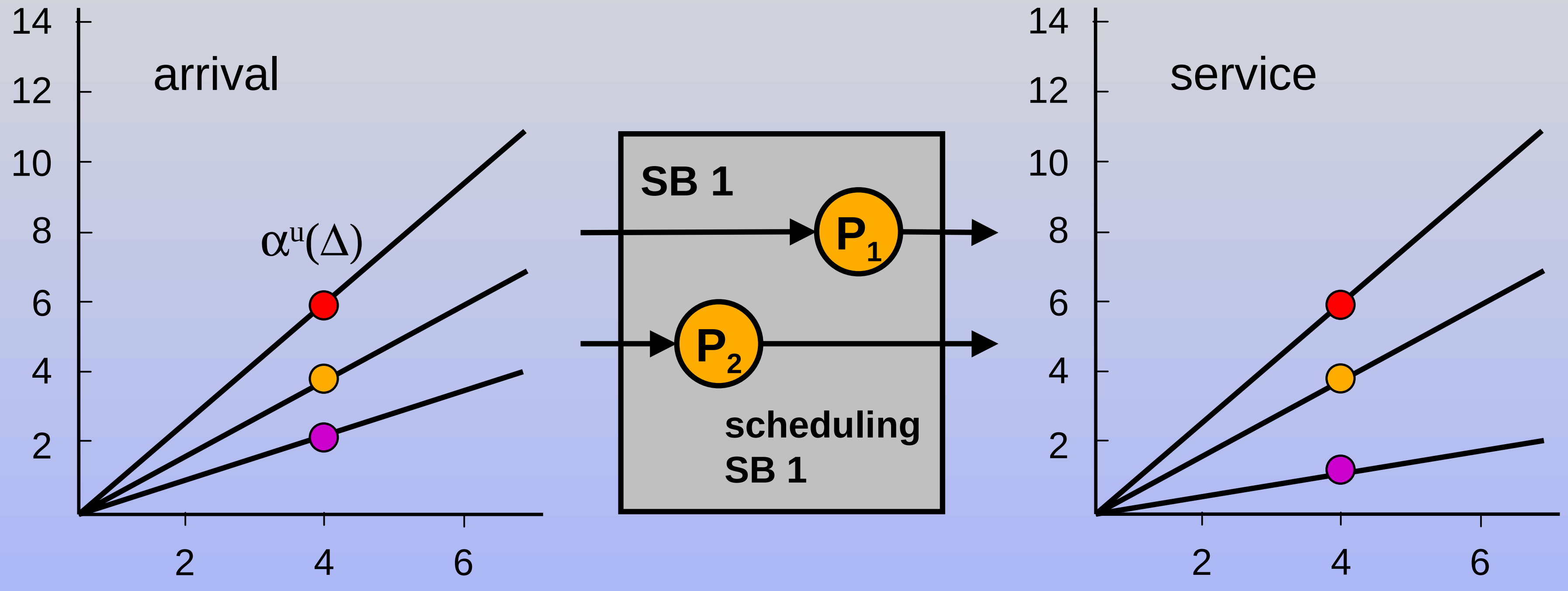
- Example: Network processor design (Thiele et al.)



- packet processing characteristics
  - dynamic load
  - interleaved flows of packets
  - flow-specific task chains
  - event bursts
  - variable data sizes

Network process generalized event model

- **Arrival/service curves (L. Thiele)**



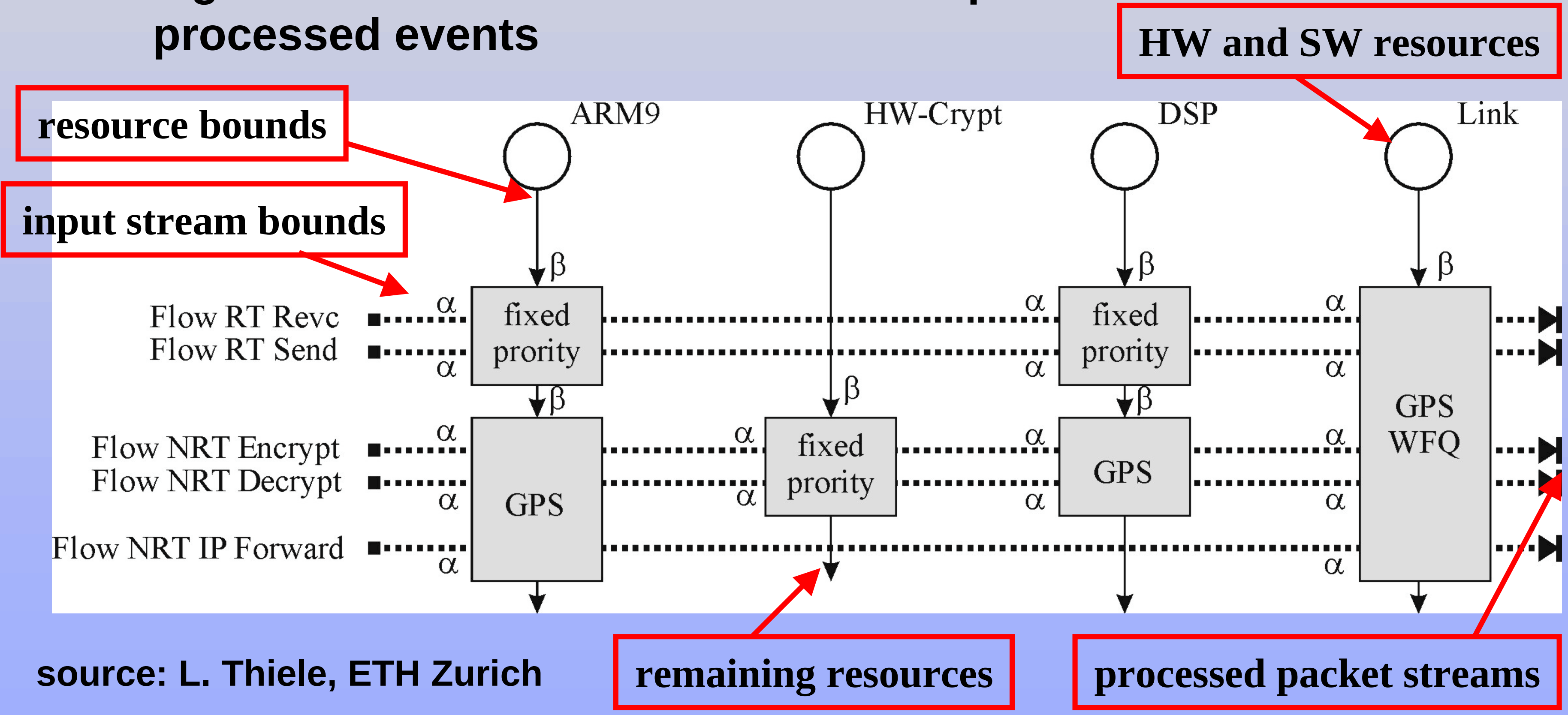
● **upper bound number of packets in any interval of length 4**

● **number of packets in time interval [0,4]**

● **lower bound number of packets in any interval of length 4**

Load analysis with interval event model

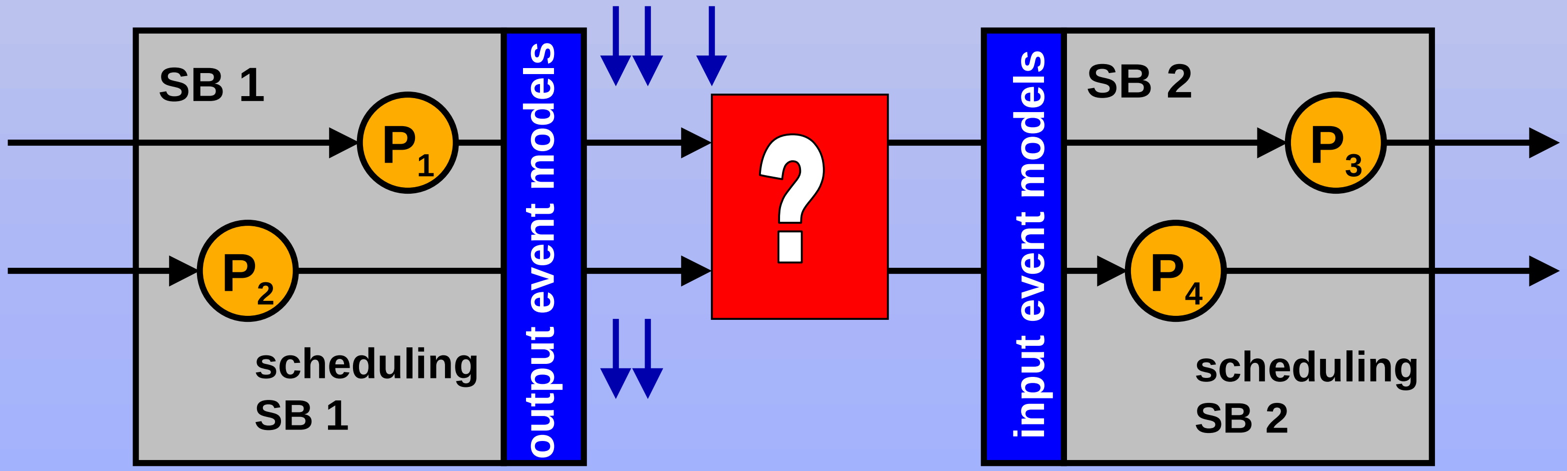
- Load analyis: Addition propagation of computation and load intervals - min/max algebra (time dependent)
- e.g. buffer size: difference between input load and processed events



source: L. Thiele, ETH Zurich

Event model adaptation

- idea:
  - leverage on the many efficient scheduling strategies available for different domains
  - utilize their corresponding analysis techniques
  - adapt their event models for combination
  - combine local results to global analysis

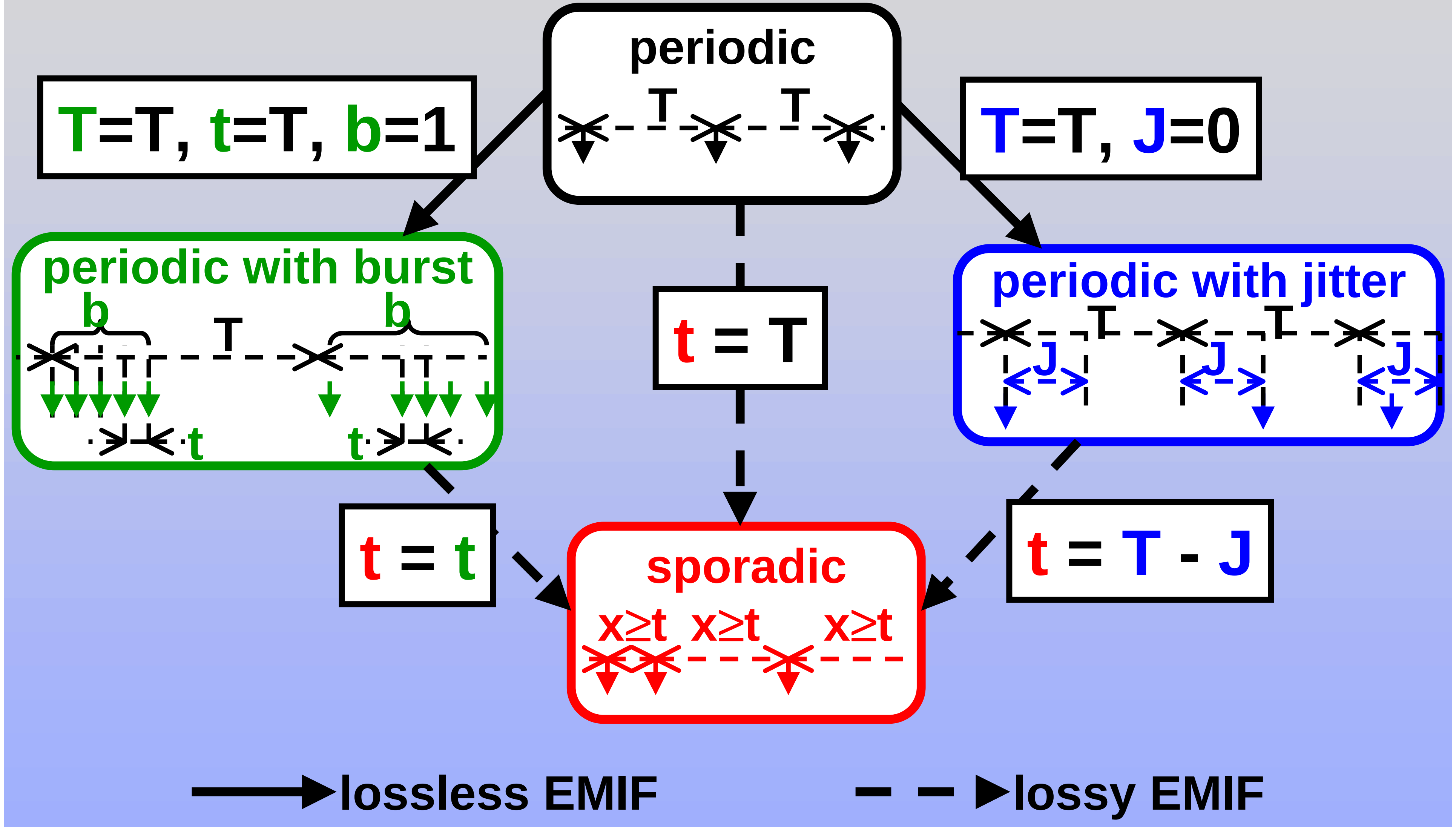


Event model adaptation - 2

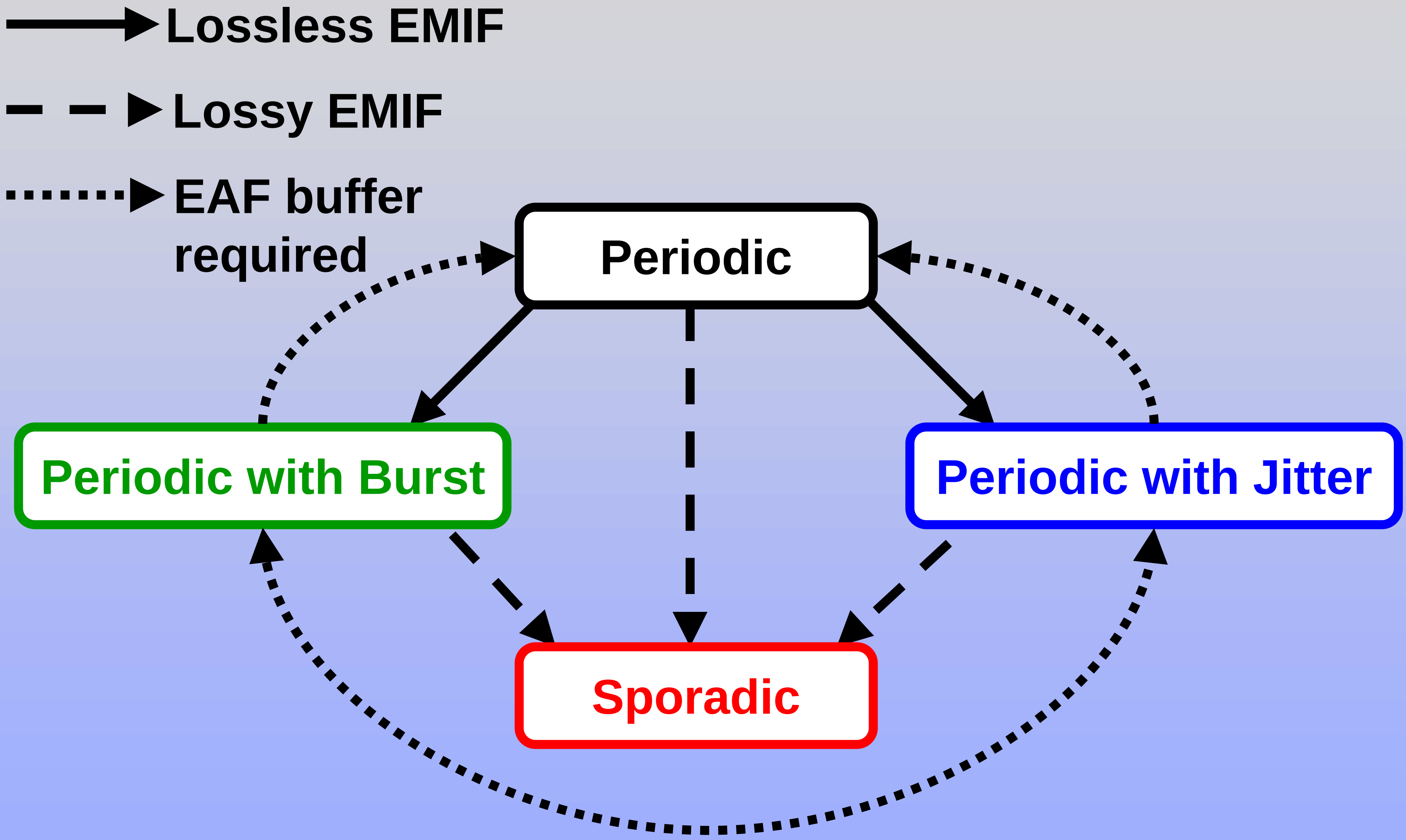
- adaptation for compatible input and output event models
  - simple mathematical transformation to adapt analysis
    Event Model InterFace (EMIF)
  - no added function, no run-time effect
- adaptation for non-compatible input and output event models
  - insert Event Adaptation Function (EAF) to transform models
  - EAF describes interface buffer function

    $\Rightarrow$ shows that buffer is required to couple subsystems

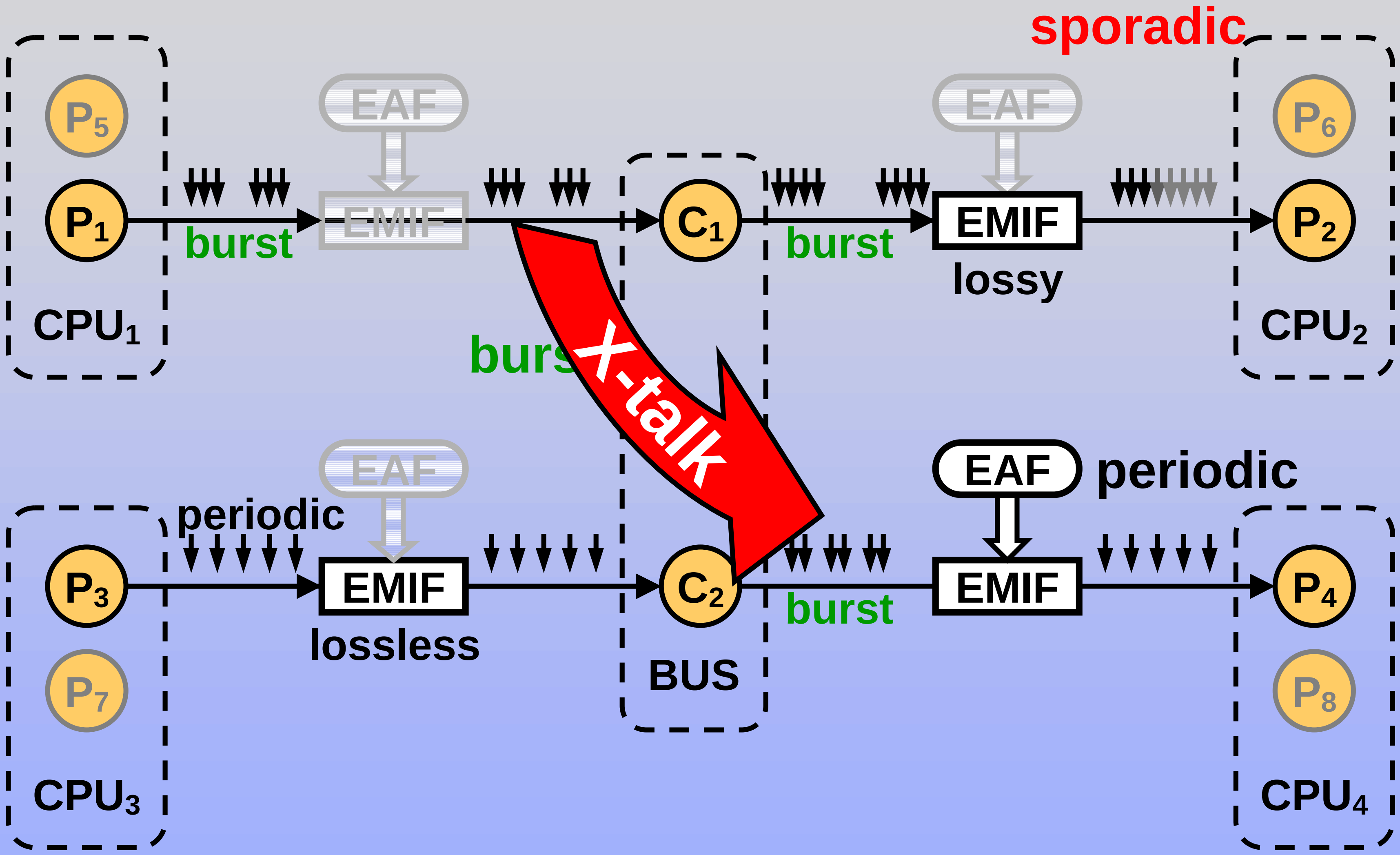
Event Model Interface Classification



Existing EMIFs and EAFs



Identification of complex interdependencies



Finally - timing and communication model

- What timing and communication model is appropriate?
  - worst case?
  - min/max (interval)?
  - typical?
  - statistics?

Timing and communication model - 2

- statistical or „typical“ timing and communication load
  - summarizes context dependent timing and communication variations
    - typical or average communication load and time can simply be accumulated
    - replaces discrete simulation by statistical analysis
  - challenges:
    - (buffer) memory overflow not fully covered
    - communication link overload not fully covered
    - critical deadlines not covered

    $\Rightarrow$ not safe

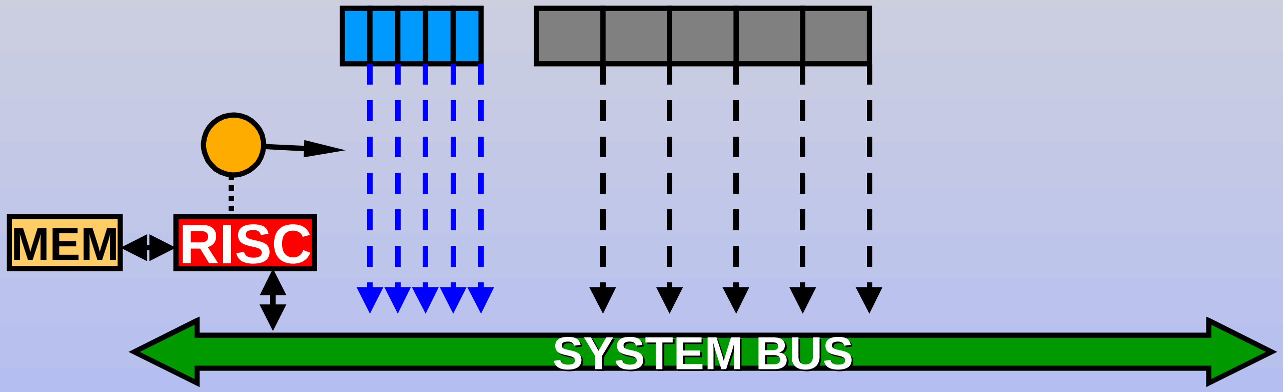
Timing and communication model - 3

- worst case timing and communication load
  - conservative
  - risk of overdesign
    - precise worst case required
    - combination with context dependent execution useful
- but: worst case timing alone is not safe, too!

Worst-case-timing-only problems

- bus load calculation

  ⇒ minimum execution time defines maximum bus load



- multiprocessor scheduling anomaly
  - fast process blocks critical process

⇒ only interval models for communication and time are safe

7 Conclusions

- **MPSoC platforms show complex run-time behavior due to optimized resource sharing**
- **implementation creates dependencies which are not reflected in the system function**
- **influence on timing and memory usage may compromize correct system behavior**
- **simulation for performance analysis is increasingly inadequate**
- **proposed systematic approach to MPSoC platform analysis exploiting knowledge from RTOS**
- **reviewed different techniques to investigate subsystem compositions**
- **techniques can also be used for estimation**

Literature

- see: www.spi-project.org

Acknowledgement

- The following persons contributed in developing these slides
  - Bettina Boettger
  - Kai Richter
  - Dirk Ziegenbein