technische universität
dortmund

fakultät für informatik
informatik 12

Embedded System Design

Embedded Systems Foundations of Cyber-Physical Systems

Peter Marwedel
TU Dortmund,
Informatik 12

2013 年 10 月 09 日

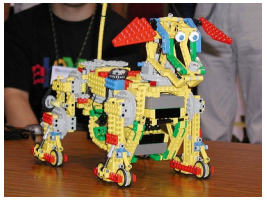
Motivation for course (1)

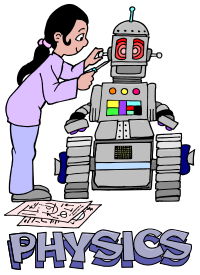
According to forecasts, future of IT characterized by terms such as

- Disappearing computer,
- Ubiquitous computing,
- Pervasive computing,
- Ambient intelligence,
- Post-PC era,
- Cyber-physical systems.

Basic technologies:

- *Embedded System technologies*
- Communication technologies





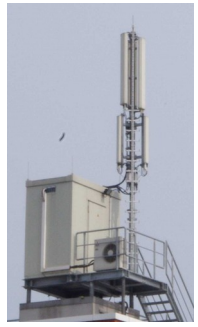
Motivation for course (2)

National Research Council Report (US)
Embedded Everywhere, 2001:

"Information technology (IT) is on the verge of another revolution. …..

networked systems of embedded computers ... have the potential to change radically the way people interact with their environment by linking together a range of devices and sensors that will allow information to be collected, shared, and processed in unprecedented ways. ...

The use … throughout society could well dwarf previous milestones in the information revolution."



Motivation for course (3)

- **The future is embedded, embedded is the future**

Embedded Systems & Cyber-Physical Systems

"Dortmund" Definition: [Peter Marwedel]
Embedded systems are information processing systems embedded into a larger product

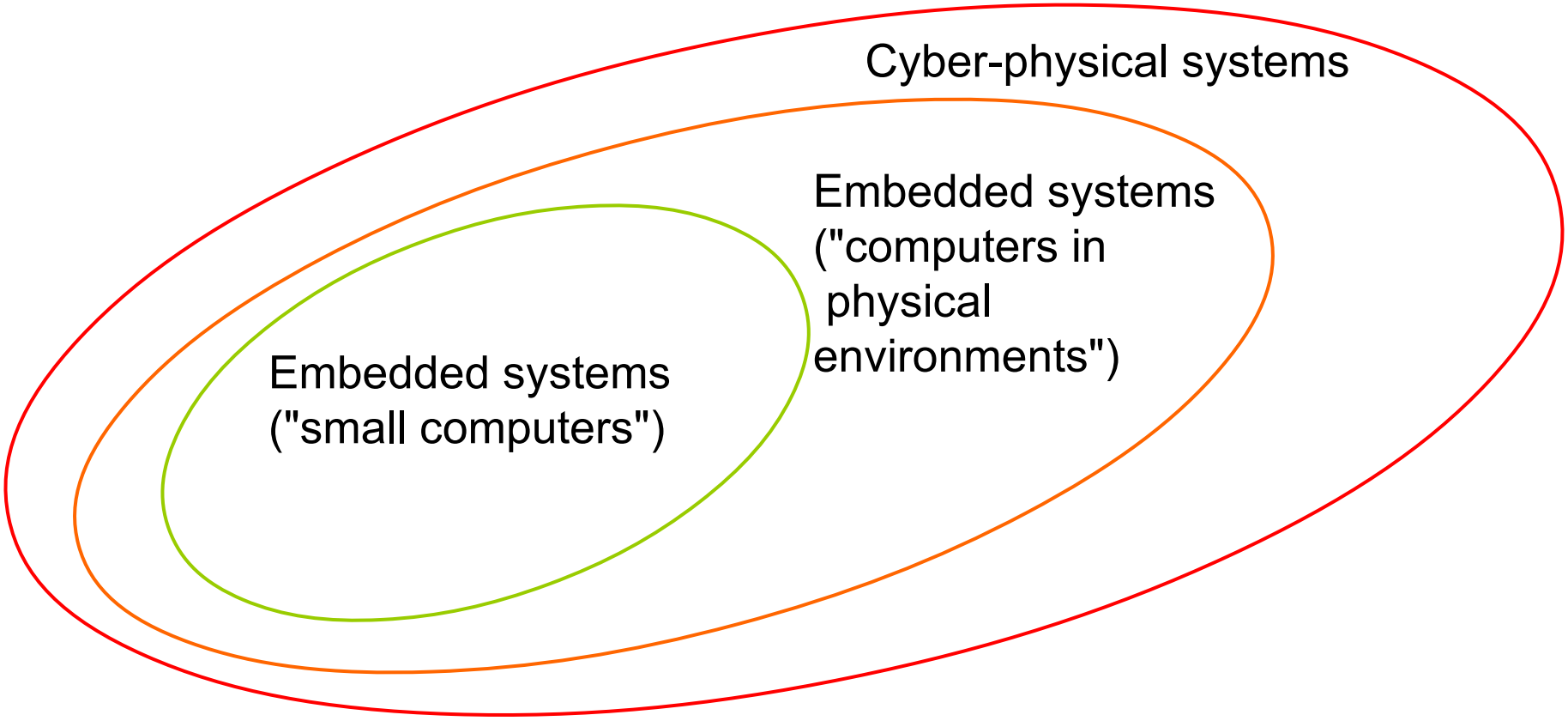
Berkeley: [Edward A. Lee]:
Embedded software is software integrated with physical processes. The technical problem is managing time and concurrency in computational systems.

Cyber-Physical (cy-phy) Systems (CPS) are integrations of computation with physical processes [Edward Lee, 2006].

Cyber-physical system (CPS) =
Embedded System (ES) + *physical environment*

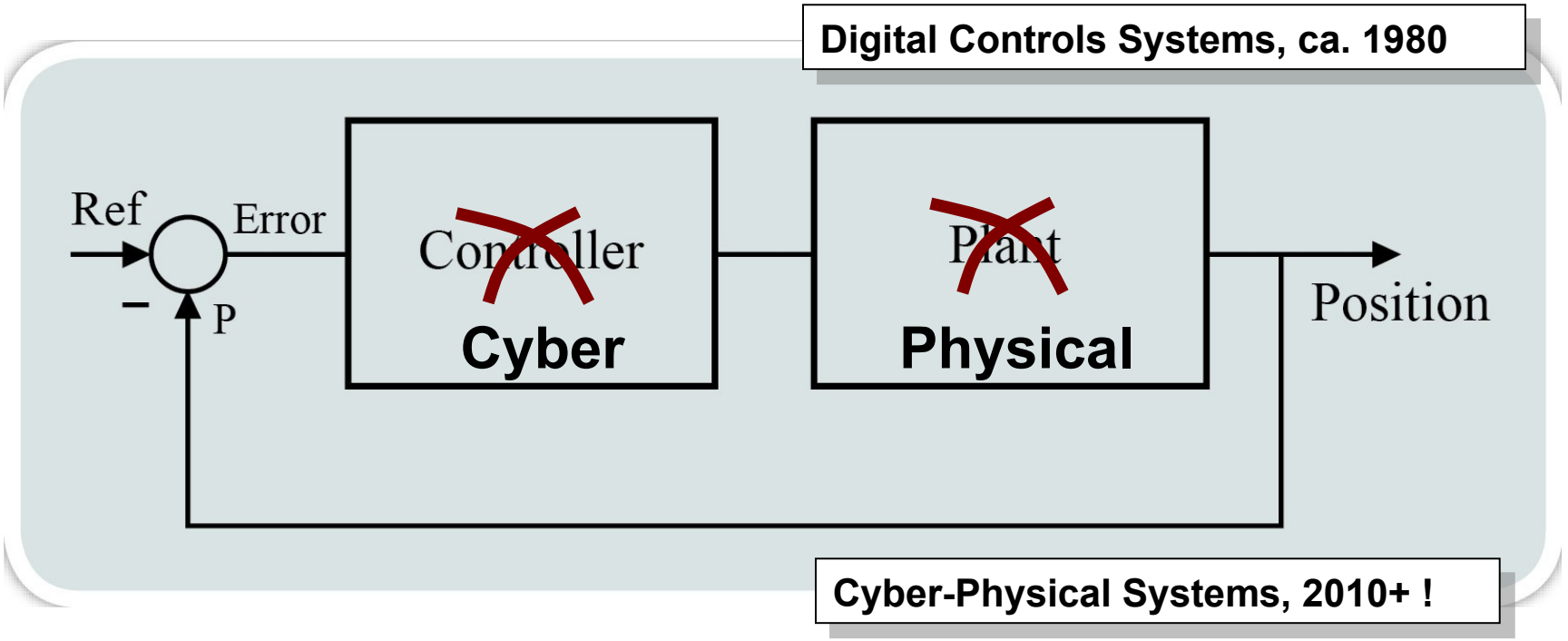
Cyber-physical systems and embedded systems

CPS = ES + physical environment



What is a *Cyber-Physical System*?

Extreme view:



Definition according to National Science Foundation (US)

Cyber-physical systems (CPS) are engineered systems that are built from and depend upon the ***synergy of computational and physical components.***

Emerging CPS will be ***coordinated, distributed****, and* ***connected****, and must be* ***robust*** *and* ***responsive****.*

The CPS of tomorrow will need to far exceed the systems of today in capability, adaptability, resiliency, safety, security, and usability.

Examples of the many CPS application areas include the ***smart electric grid, smart transportation, smart buildings, smart medical technologies, next-generation air traffic management, and advanced manufacturing****.*

CPS: Integration of Cyber and Physics

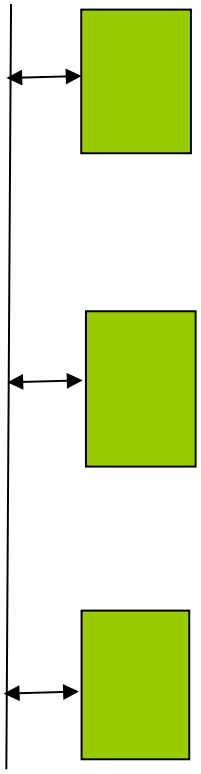
Cyber

Physics

CPS

Definition according to akatech

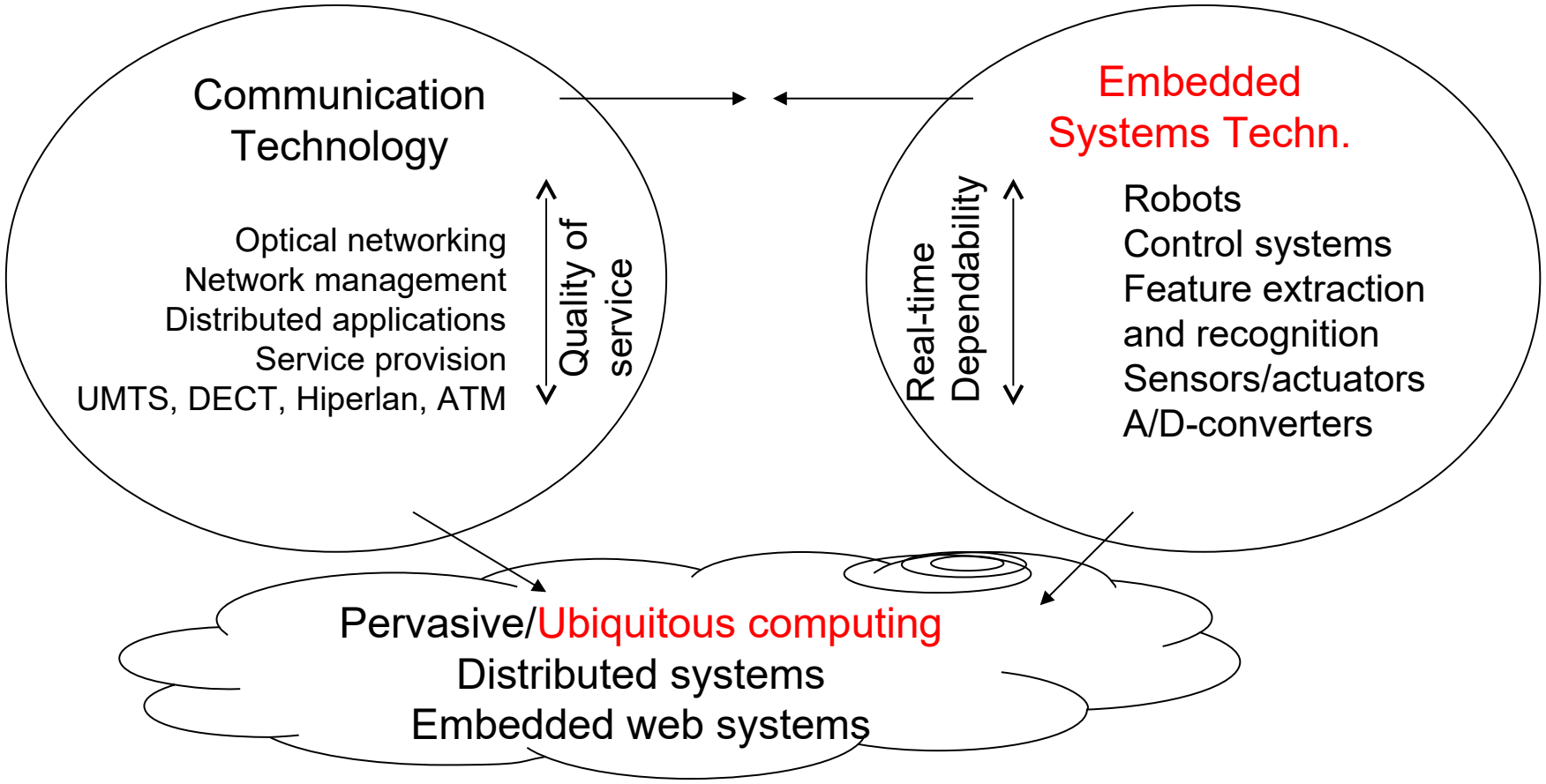
The physical world and the virtual world – or cyber-space – are merging; cyber-physical systems are developing. Future cyber-physical systems will contribute to security, efficiency, comfort and health systems as never before, and as a result, they will contribute to solving key challenges of our society, such as the aging population, limited resources, mobility, or energy transition.



[Akatech: Cyber-Physical Systems. Driving force for innovation in mobility, health, energy and production http://www.acatech.de/de/publikationen/stellungnahmen/kooperationen/detail/artikel/cyber-physical-systems-innovationsmotor-fuer-mobilitaet-gesundheit-energie-und-produktion.html]

Extending the motivation: Embedded systems and ubiquitous computing

Ubiquitous computing: Information anytime, anywhere. Embedded systems provide fundamental technology.



Application areas and examples

Application area Automotive electronics: clearly cyber-physical

Functions by embedded processing:

- ABS: Anti-lock braking systems
- ESP: Electronic stability control
- Airbags
- Efficient automatic gearboxes
- Theft prevention with smart keys
- Blind-angle alert systems
- ... etc ...

- Multiple networks
- Multiple networked processors

Application area avionics: also cyber-physical

- flight control systems,
- anti-collision systems,
- pilot information systems,
- power supply system,
- flap control system,
- entertainment system,
- …

Dependability is of outmost importance.

More application areas:

- railroad

- water ways

Dependability is of outmost importance.

Forestry machines: cyber-physical

Networked computer system

- Controlling arms & tools
- Navigating the forest
- Recording the trees harvested
- Crucial to efficient work

“Tough enough to be out in the woods”

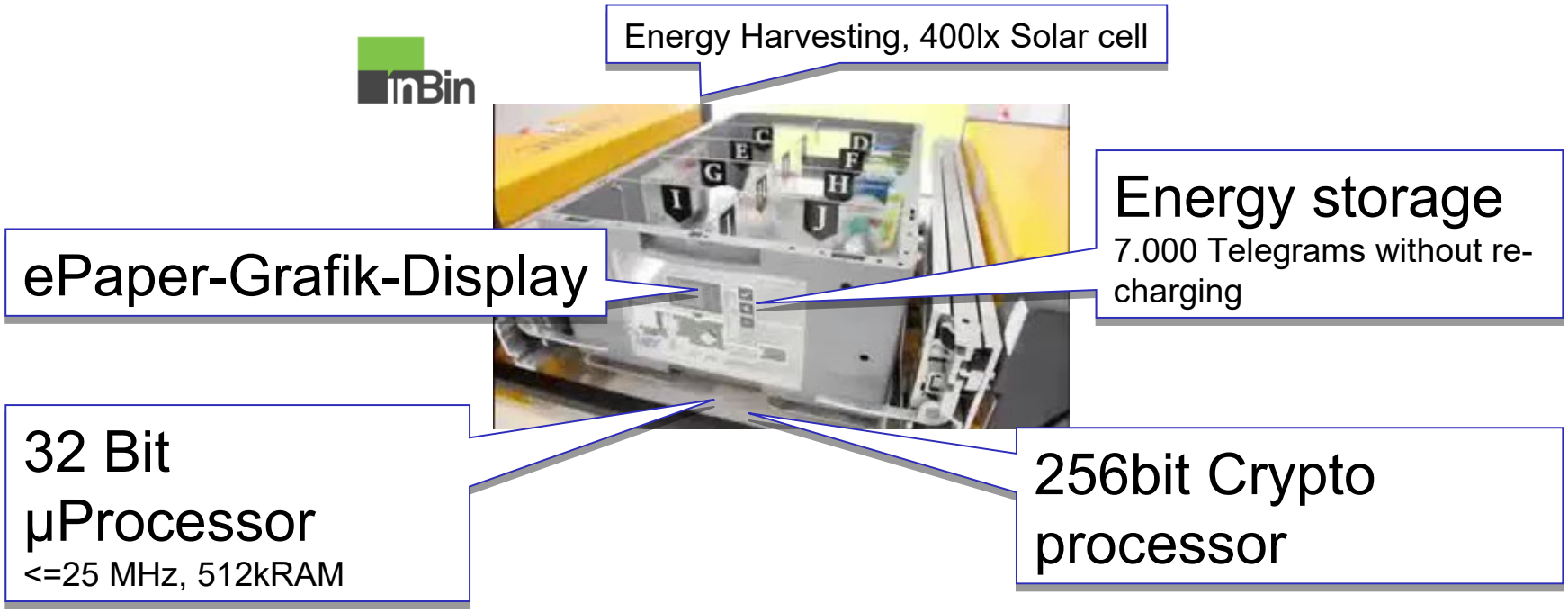
Logistics

Applications of embedded/cyber-physical system technology to logistics:

- Radio frequency identification (RFID) technology provides easy identification of each and every object, worldwide.
- Mobile communication allows unprecedented interaction.
- The need of meeting real-time constraints and scheduling are linking embedded systems and logistics.
- The same is true of energy minimization issues

Internet of Things

Internet of things and services



Innovationspartner:
Würth Industrie Services GmbH
Debrunner Koenig Management AG

Fabrication

Production resources are self-configuring and distributed *social machines*

Industry 4.0

© Fraunhofer IML, Dortmund

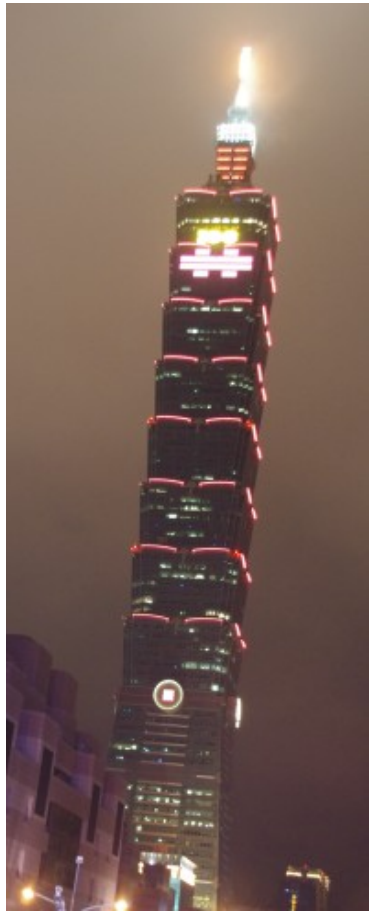
Structural safety

Sensors + data analysis

Möhne lake dam

Kilauea, Hawaii

Bridge at Vancouver



Taipeh 101

Smart Home

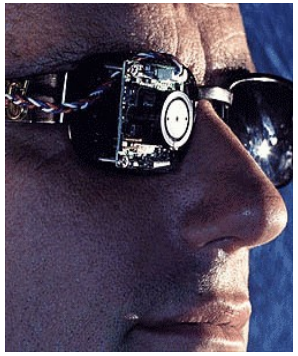
- Zero energy building, generates as much energy as it consumes
- Provides safety and security
- Supports owners
- Provides maximum comfort
- Ambient assisted living

© P. Marwedel

Medical systems: cyber-physical

For example

- Artificial eye
  several approaches, e.g.:
  - camera attached to glasses
  - computer worn at belt
  - output directly connected to the brain
  - “pioneering work by William Dobelle”.
    Previously at [www.dobelle.com]

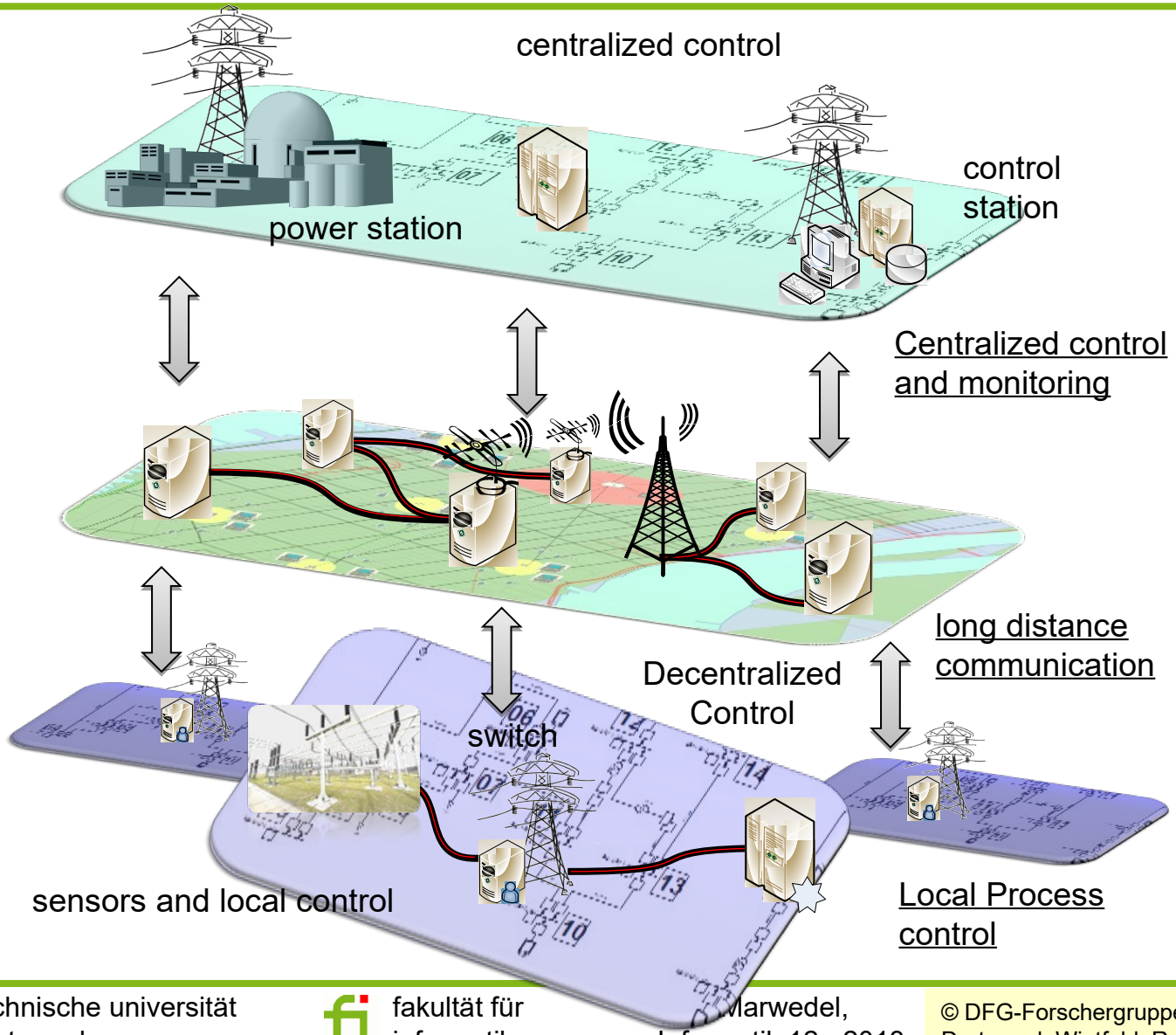


- Translation into sound,
  claiming much better resolution.
  [http://www.seeingwithsound.com/etumble.htm]

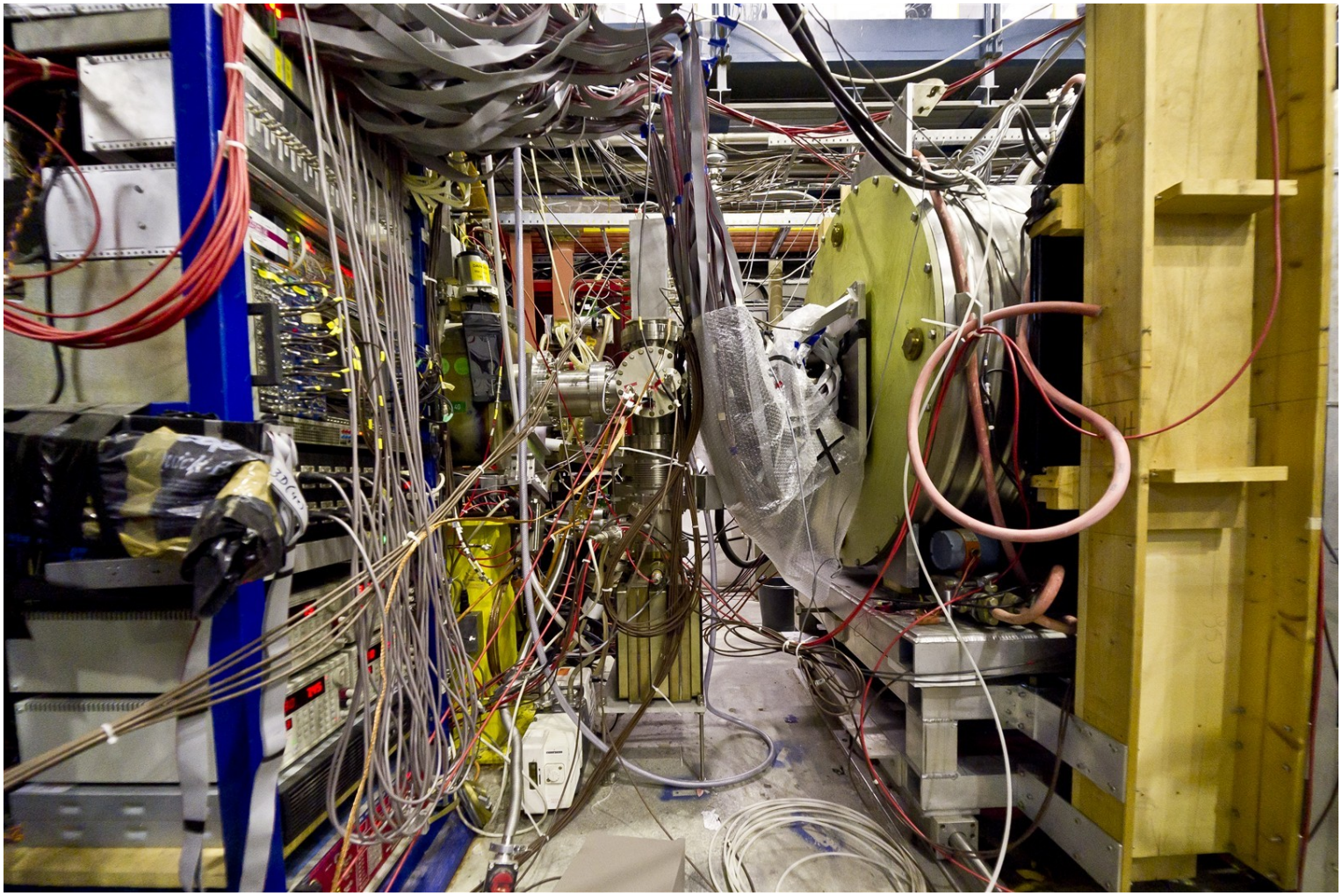
Smart Medicine

- Diagnosis
- Support of therapy
- evaluation
- risk analysis
- Information about patients

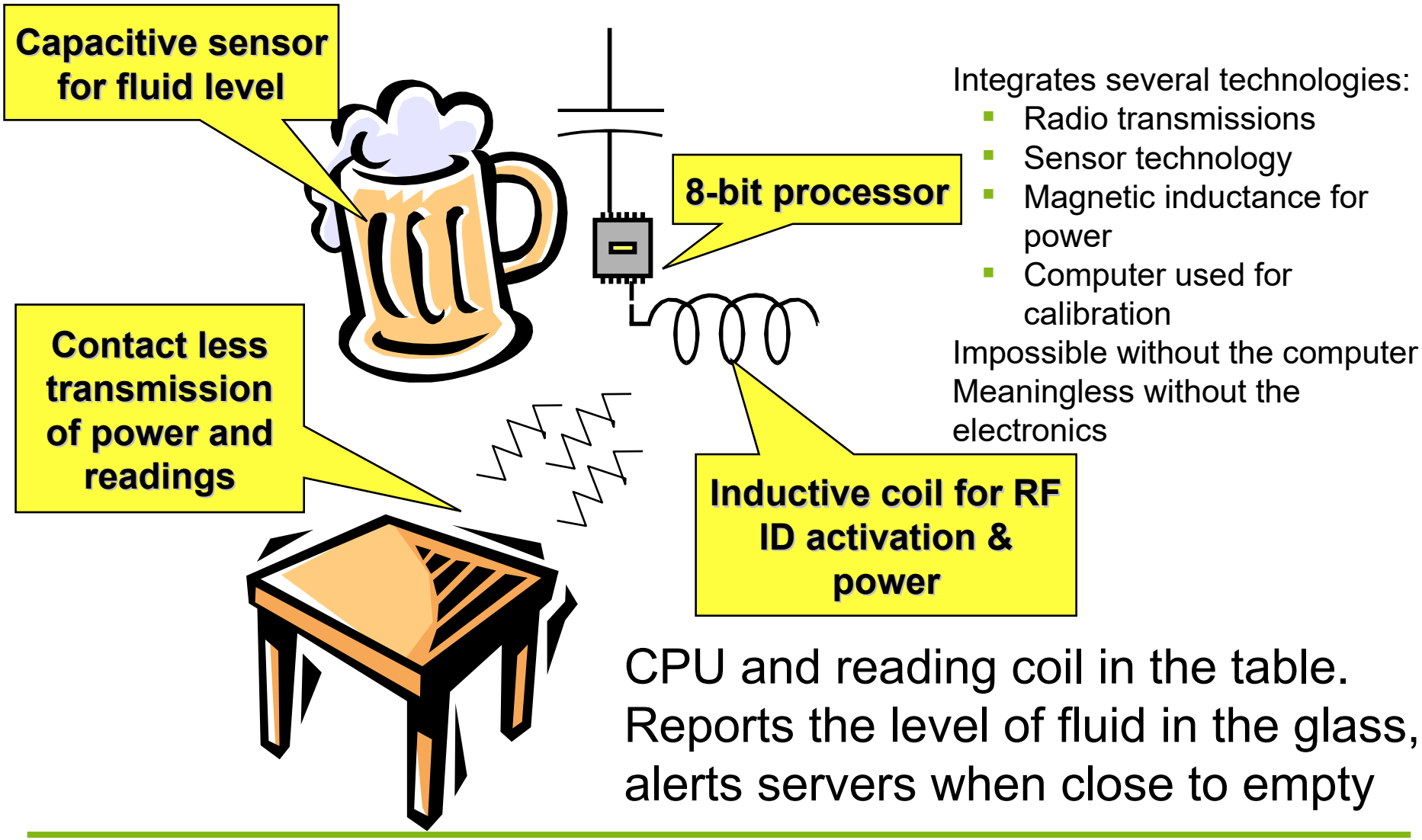
Smart Grid



Integration of Physics and Cyber in Physical Experiments



Smart Beer Glass



Integrates several technologies:

- Radio transmissions
- Sensor technology
- Magnetic inductance for power
- Computer used for calibration

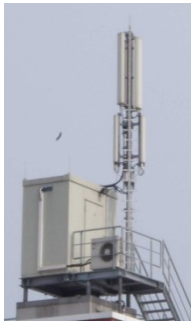
Impossible without the computer
Meaningless without the electronics

CPU and reading coil in the table. Reports the level of fluid in the glass, alerts servers when close to empty

More application areas

- Telecommunication
- Consumer electronics
- Robotics
- Public safety
- Military systems

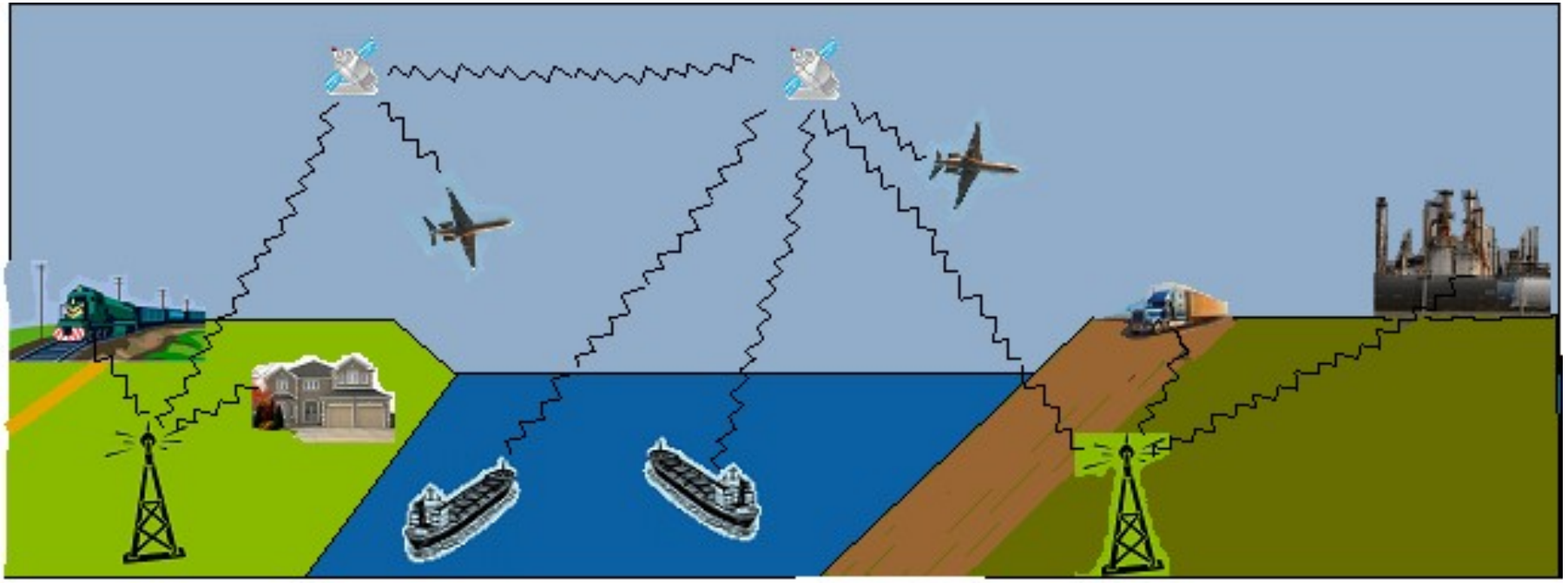






Mostly cyber-physical

Connecting previously isolated systems



Educational concept

Broad set of topics

1. Introduction
2. Specification and modeling
3. CPS/ES hardware
4. CPS/ES system software
5. Evaluation
6. Mapping of applications to execution platforms
7. Optimizations
8. Test

Slides

- Slides are available at:
  - http://ls12-www.cs.tu-dortmund.de/~marwedel/es-book
- Master Format: Powerpoint (2010 -new-)
- Derived Format: PDF

Summary

- A look at the future of IT
- Definition: embedded & cyber-physical (cy-phy) systems
- Growing importance of embedded & cy-phy systems
- Application areas & examples
- Curriculum

technische universität
dortmund

fakultät für informatik
informatik 12

Embedded System Design

Embedded Systems Foundations of Cyber-Physical Systems

Peter Marwedel
TU Dortmund,
Informatik 12

2013 年 10 月 09 日

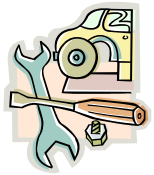
Common characteristics

Dependability

- CPS/ES must be **dependable**,

  - **Reliability $R(t)$ =** probability of system working correctly provided that is was working at t=0

  - **Maintainability $M(d)$ =** probability of system working correctly d time units after error occurred.



  - **Availability $A(t)$**: probability of system working at time t
  - **Safety**: no harm to be caused

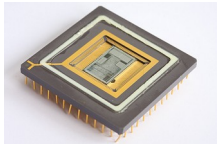
  - **Security**: confidential and authentic communication

Even perfectly designed systems can fail if the assumptions about the workload and possible errors turn out to be wrong.

Making the system dependable must not be an after-thought, it must be considered from the very beginning

Efficiency

- CPS & ES must be **efficient**
  - Code-size efficient
    (especially for systems on a chip)

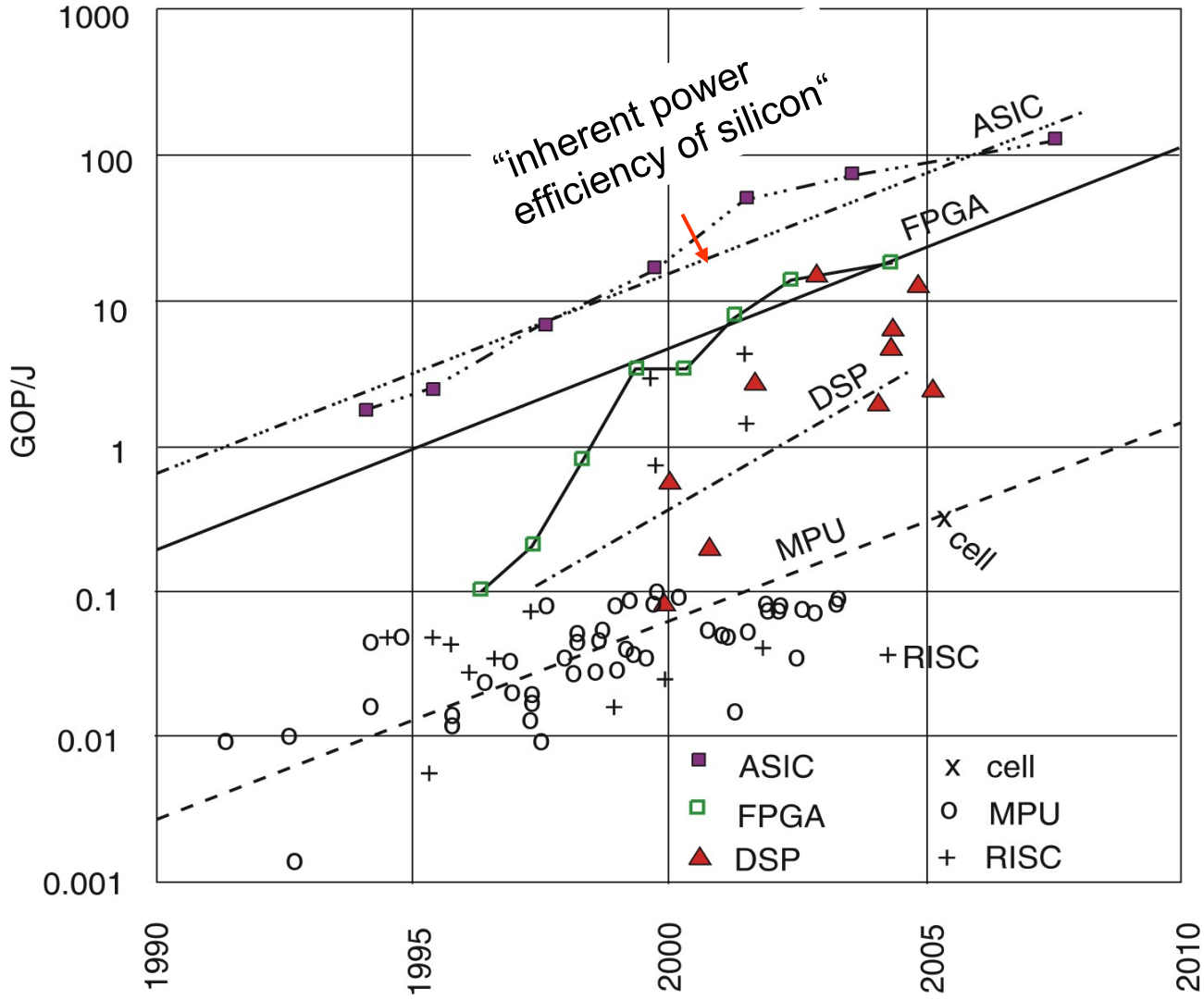


  - Run-time efficient

  - Weight efficient

  - Cost efficient
  - Energy efficient

Importance of Energy Efficiency

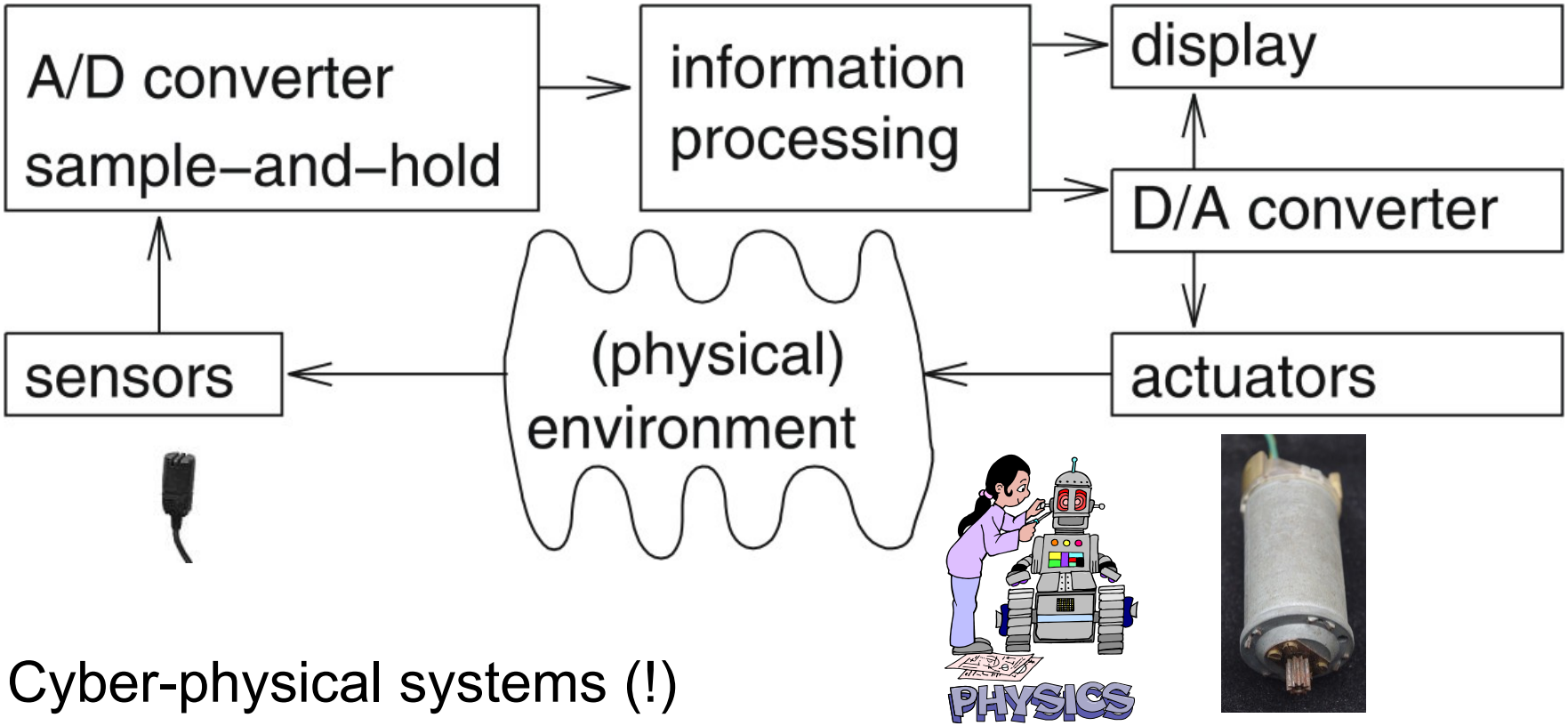


Efficient software design needed, otherwise, the price for software flexibility cannot be paid.

© Hugo De Man, IMEC, Philips, 2007

CPS & ES Hardware

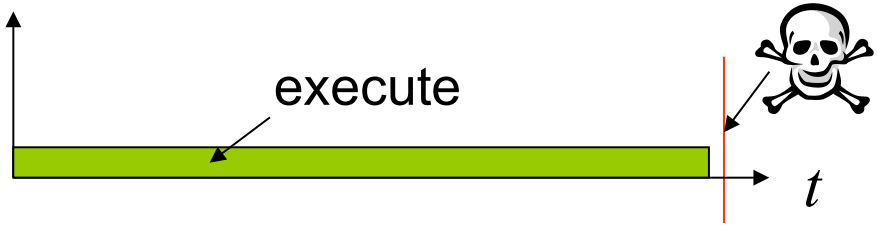
CPS & ES hardware is frequently used in a loop (***"hardware in a loop"***):

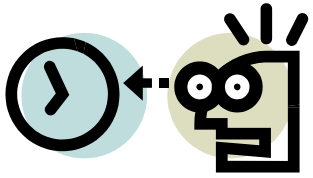


Cyber-physical systems (!)

Real-time constraints

- CPS must meet **real-time constraints**
  - A real-time system must react to stimuli from the controlled object (or the operator) within the time interval **dictated** by the environment.





  - **"A real-time constraint is called hard, if not meeting that constraint could result in a catastrophe"** [Kopetz, 1997].
  - All other time-constraints are called **soft**.
  - A guaranteed system response has to be explained without statistical arguments [Kopetz, 1997].

Typical Misconceptions

"Real time" is performance engineering/tuning.

- Timeliness is more important in real-time systems.

Real-time computing is equivalent to fast computing.

- Real-time computing means predictable and reliable computing.

Advances in supercomputing hardware will take care of real-time requirements.

- Buying a "faster" processor may result in timeliness violation.

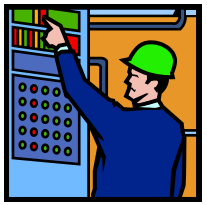
Real-Time Systems & CPS

CPS, ES and Real-Time Systems synonymous?

- For some embedded systems, real-time behavior is less important (smart phones)
- For CPS, real-time behavior is essential, hence RTS ≅ CPS
- CPS models also include a model of the physical system

Reactive & hybrid systems

- Typically, CPS are **reactive systems**:
  "**A reactive system is one which is in continual interaction with is environment and executes at a pace determined by that environment**"
  [Bergé, 1995]



  Behavior depends on input **and current state**.
  ☞ automata model appropriate,
  model of computable functions inappropriate.

- **Hybrid systems**
  (analog + digital parts).

Dedicated systems

- **Dedicated** towards a certain **application**
  Knowledge about behavior at design time can be used to minimize resources and to maximize robustness

- **Dedicated user interface**
  (no mouse, keyboard and screen)

- Situation is slowly changing here: systems become less dedicated

Security

- Defending against
  - Cyber crime („Annual U.S. Cybercrime Costs Estimated at $100 Billion; …[Wall Street Journal, 22.7.2013])
  - Cyber attacks (☞ Stuxnet)
  - Cyber terrorism
  - Cyber war (Cyber-Pearl-Harbor [Spiegel Online, 13.5.2013])

- Connectivity increases threats
  - entire production chains can be affected
  - local islands provide some encapsulation, but contradict idea of global connectedness

Dynamics

Frequent change of environment

Underrepresented in teaching

- CPS & ES are **underrepresented in teaching** and public discussions:
  "*Embedded chips aren't hyped in TV and magazine ads ...*" [Mary Ryan, EEDesign, 1995]

Not every CPS & ES has all of the above characteristics.

Def.: Information processing systems having most of the above characteristics are called embedded systems.

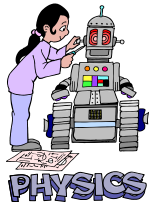
Course on embedded systems foundations of CPS makes sense because of the number of common characteristics.

Characteristics lead to corresponding challenges

- Dependability

- Efficiency
  - In particular: Energy efficiency

- Hardware properties, physical environment



- Meeting real time requirements

- …

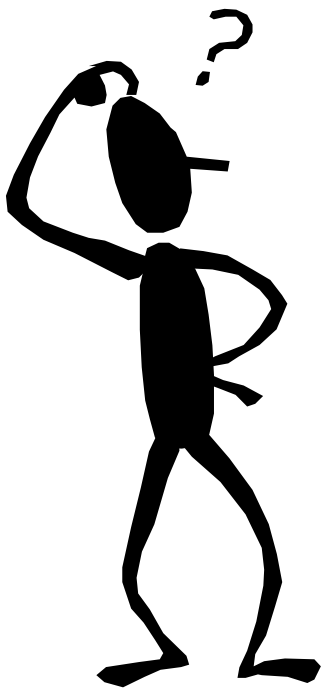
Challenges for implementation in hardware

- Early embedded systems frequently implemented in hardware (boards)
- Mask cost for specialized application specific integrated circuits (ASICs) becomes very expensive (M$ range, technology-dependent)
- Lack of flexibility (changing standards).

☞ Trend towards implementation in software (or possibly FPGAs, see chapter 3)

Challenges for implementation in software

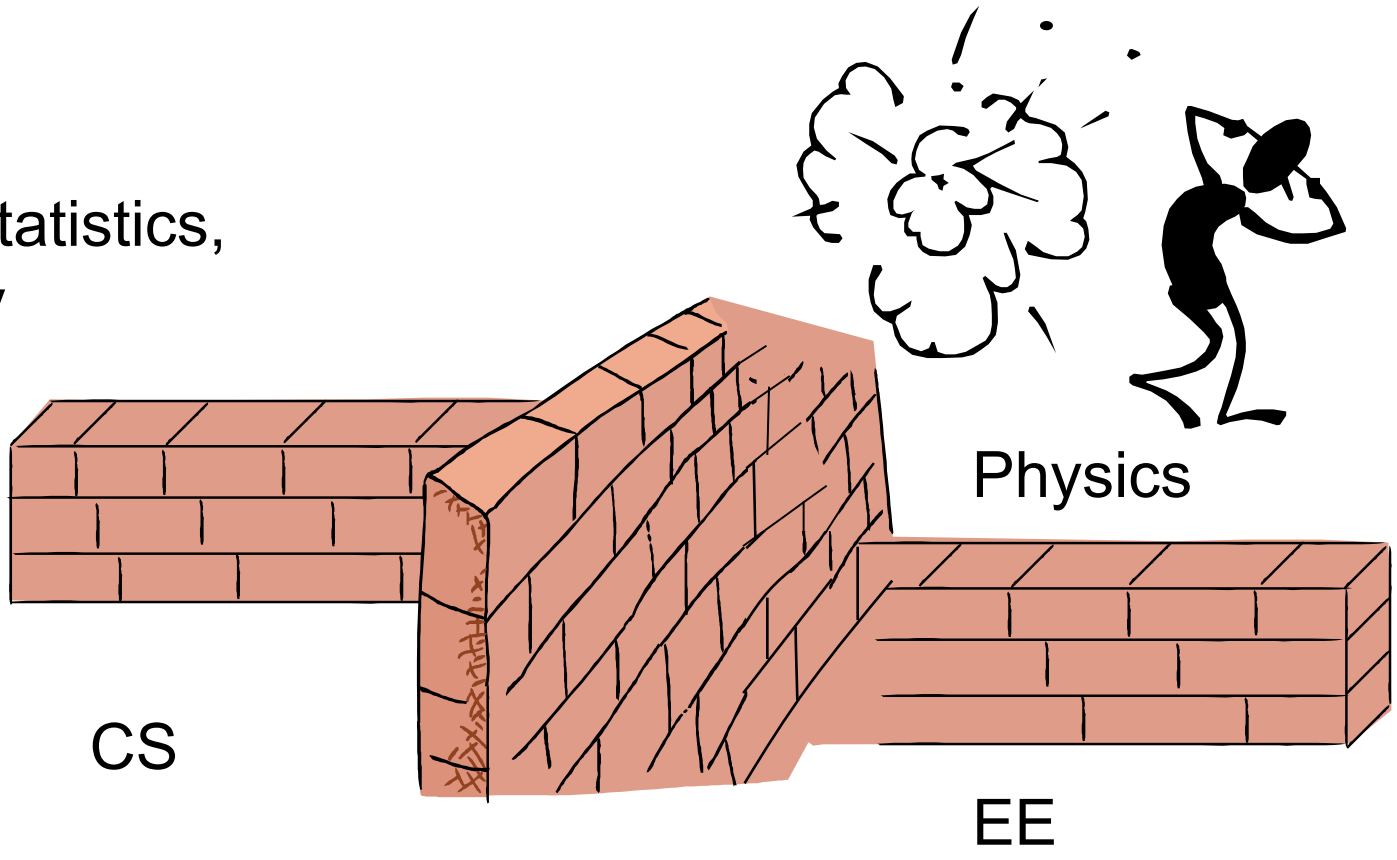
If CPS/ES will be implemented mostly in software, then why don‘t we just use what software engineers have come up with?



It is not sufficient to consider CPS/ES as a special case of SW engineering

Knowledge from many areas must be available,
Walls between disciplines must be torn down

medicine, statistics,
ME, biology



Challenges for CPS/ES Software

- Dynamic environments
- Capture the required behaviour!
- Validate specifications
- Efficient translation of specifications into implementations!
- How can we check that we meet real-time constraints?
- How do we validate embedded real-time software? (large volumes of data, testing may be safety-critical)

Software complexity is a challenge

Software in a TV set

- Source 1*:

Year	Size
1965	0
1979	1 kB
1990	64 kB
2000	2 MB

- Source 2°: 10x per 6-7 years

Year	Size
1986	10 KB
1992	100 kB
1998	1 MB
2008	15 MB

☞ Exponential increase in software complexity

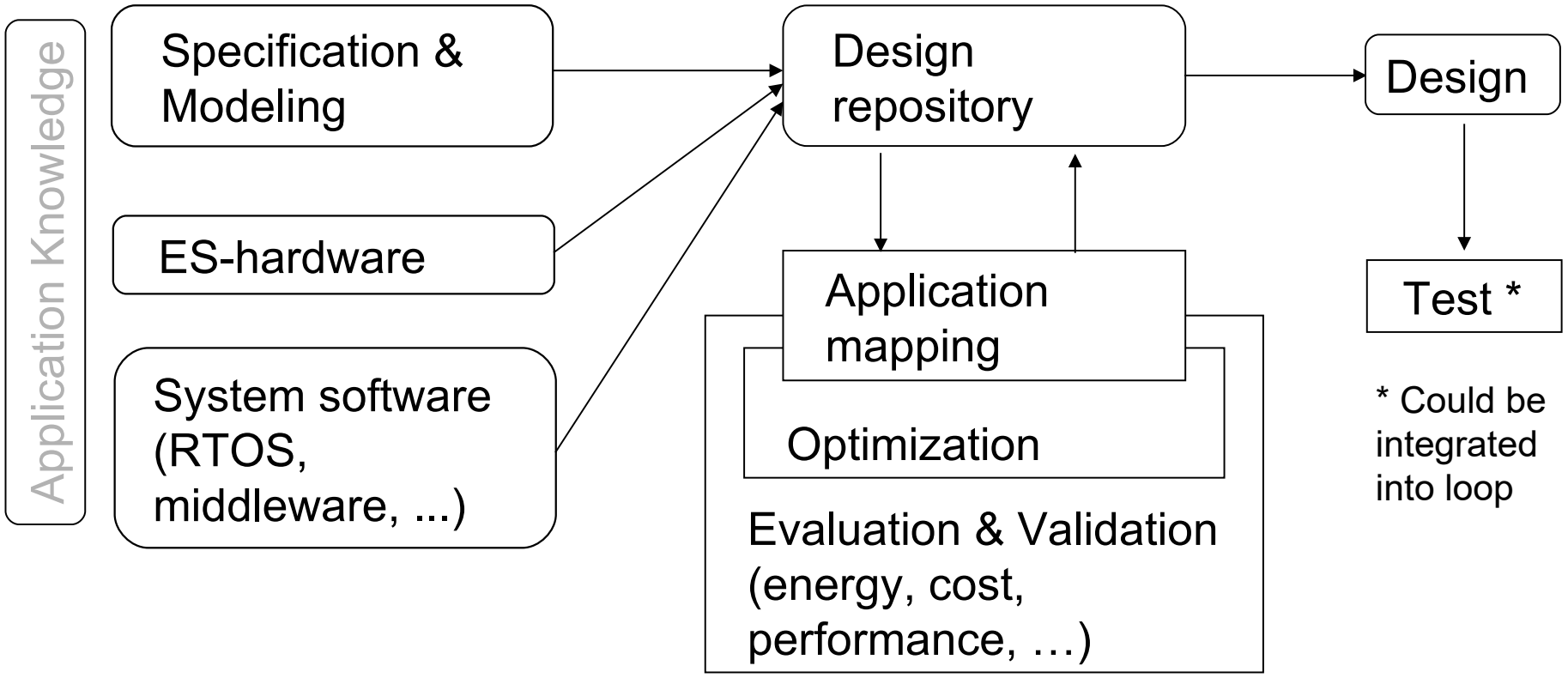
- *... > 70% of the development cost for complex systems such as automotive electronics and communication systems are due to software development*
  [A. Sangiovanni-Vincentelli, 1999]

* Rob van Ommering, COPA Tutorial, as cited by: Gerrit Müller: Opportunities and challenges in embedded systems, *Eindhoven Embedded Systems Institute*, 2004

° R. Kommeren, P. Parviainen: Philips experiences in global distributed software development, *Empir Software Eng*. (2007) 12:647-660

Design flows

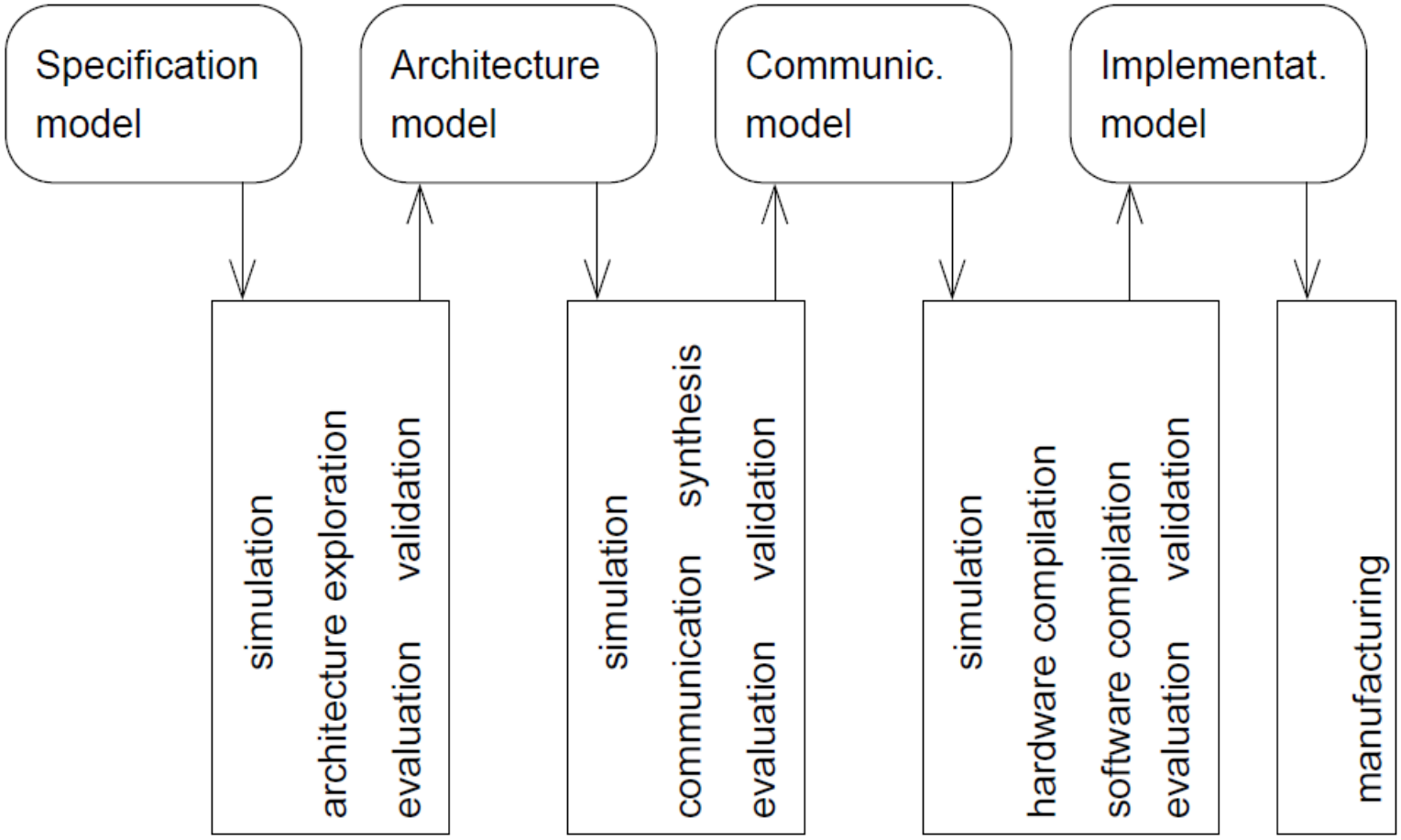
Design flow – Iterative design



Generic loop: tool chains differ in the number and type of iterations

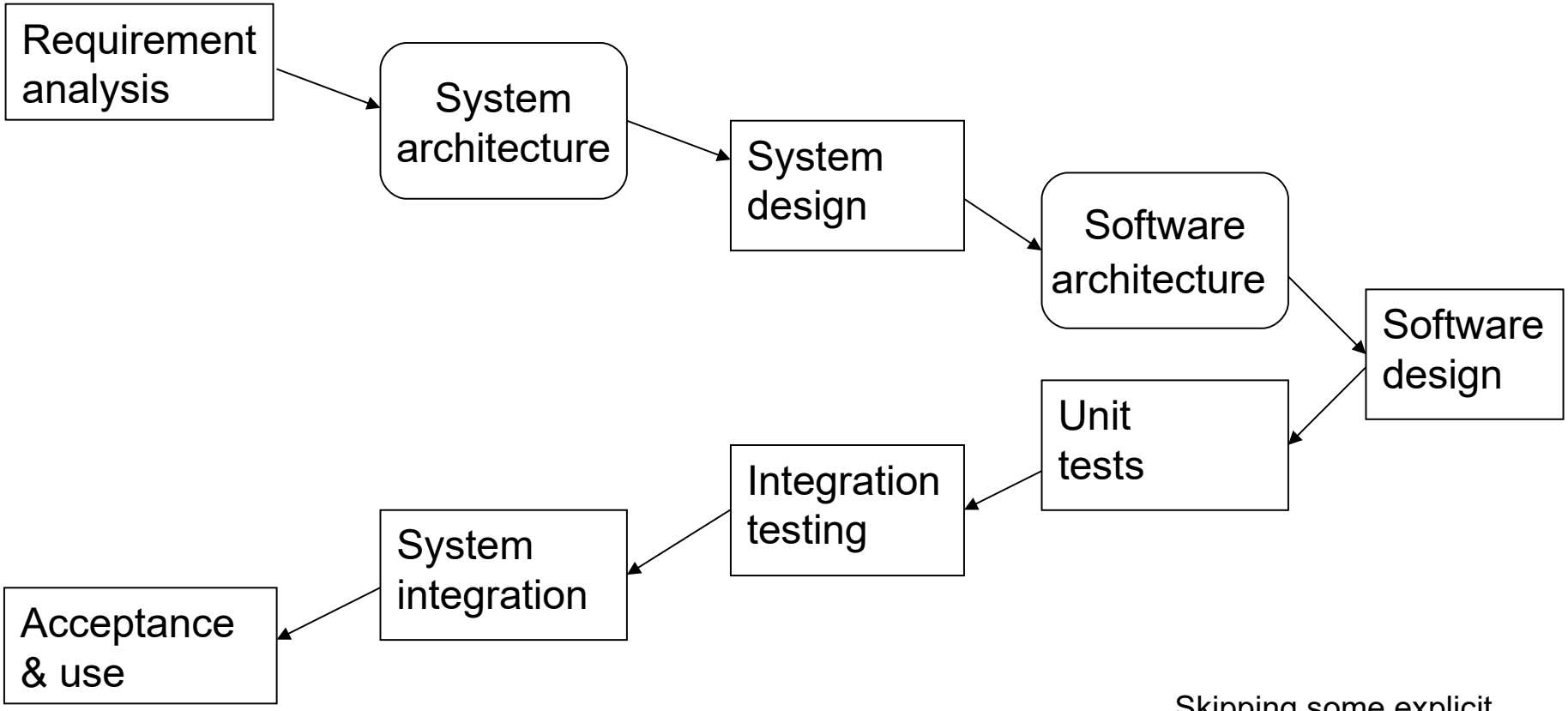
Iterative design (1): - After unrolling loop -

Example: SpecC tools



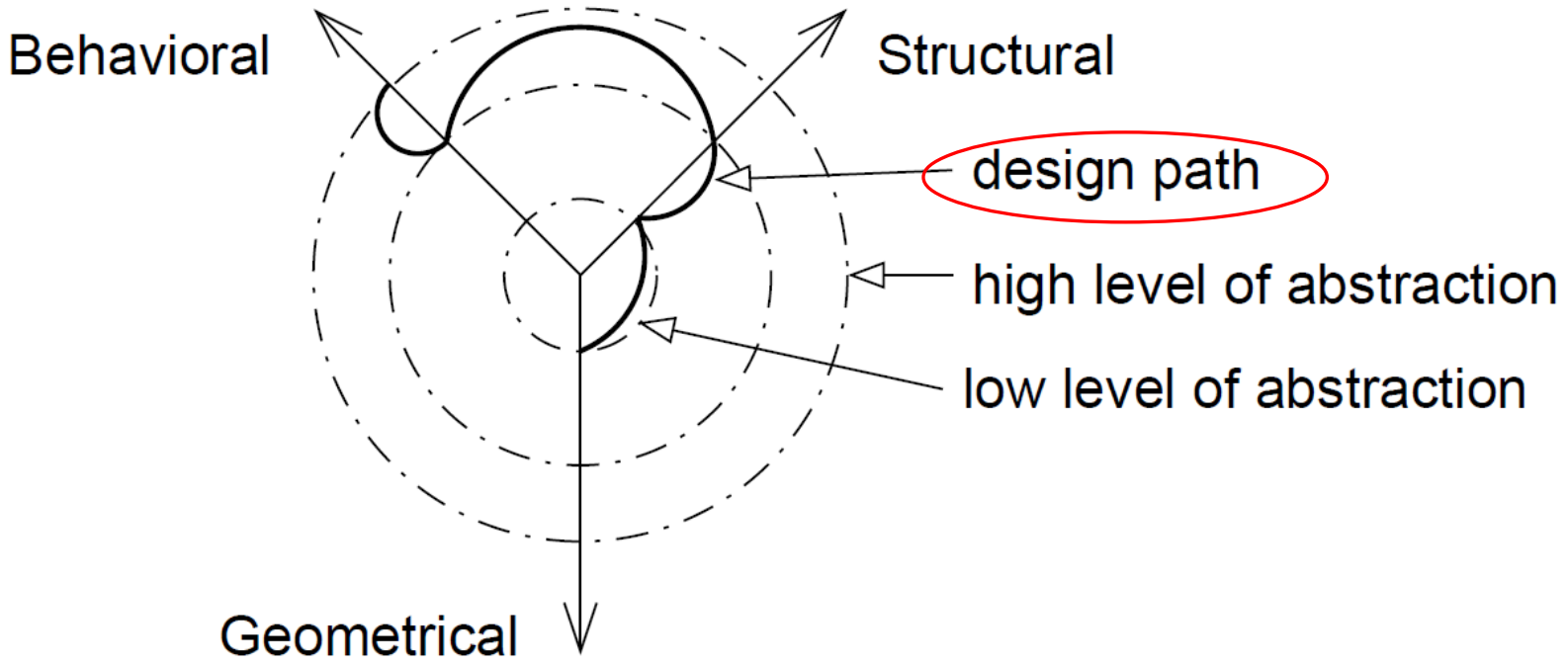
Iterative design (2): - After unrolling loop -

Example: V-model



Skipping some explicit repository updates ..

Iterative design (3): - Gajski‘s Y-chart -



Summary

- Common characteristics
- Challenges (resulting from common characteristics)
- Design Flows

technische universität
dortmund

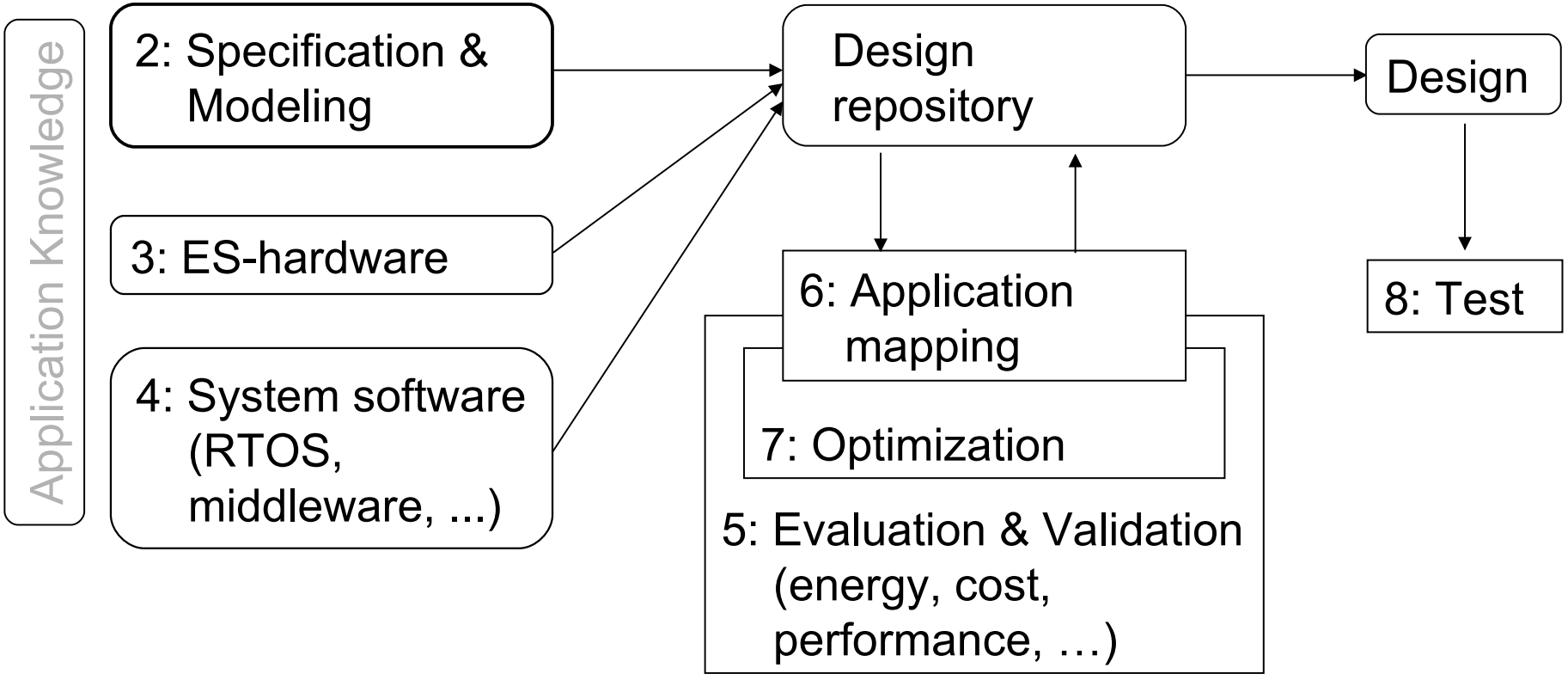
fakultät für informatik
informatik 12

Specifications and Modeling

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 10 月 17 日

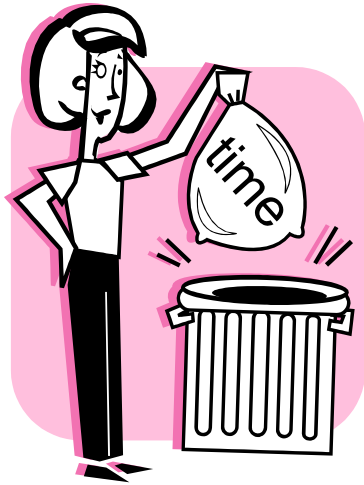
Hypothetical design flow



Numbers denote sequence of chapters

Motivation for considering specs & models

- Why considering specs and models in detail?
- If something is wrong with the specs,
  then it will be difficult to get the design right,
  potentially wasting a lot of time.
- Typically, we work with **models** of the **system under design** (SUD)



☞ What is a *model* anyway?

Models

Definition: *A model is a simplification of another entity, which can be a physical thing or another model. The model contains exactly those characteristics and properties of the modeled entity that are relevant for a given task. A model is minimal with respect to a task if it does not contain any other characteristics than those relevant for the task*.

[Jantsch, 2004]

Which requirements do we have for our models?

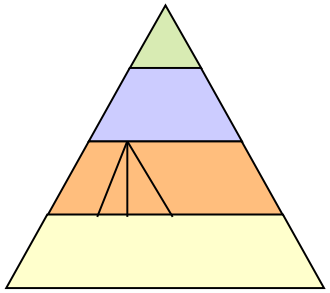
Requirements for specification & modeling techniques: 1. Hierarchy

Hierarchy

Humans not capable to understand systems containing more than ~5 objects.

Most actual systems require more objects

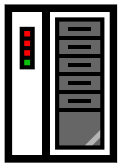
☞ Hierarchy (+ abstraction)



proc
  proc
    proc

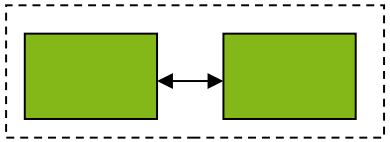
- Behavioral hierarchy
  Examples: states, processes, procedures.

- Structural hierarchy
  Examples: processors, racks, printed circuit boards



Requirements for specification & modeling techniques: 2. Component-based design

- Systems must be designed from components
- Must be “easy” to derive behavior from behavior of subsystems



☞ Work of Sifakis, Thiele, Ernst, …

- Concurrency
- Synchronization and communication

Requirements for specification & modeling techniques: 3. Timing (1)

- Timing behavior
  Essential for embedded and cy-phy systems!

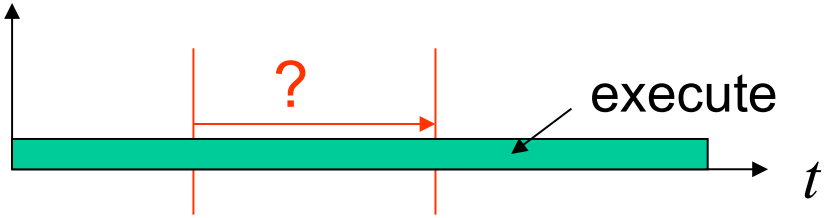
  - Additional information welcome
    (periods, dependences, scenarios, use cases)
  - Also, the speed of the underlying platform must be known
  - Far-reaching consequences for design processes!

"The lack of timing in the core abstraction (of computer science) *is a flaw, from the perspective of embedded software"*
[Lee, 2005]

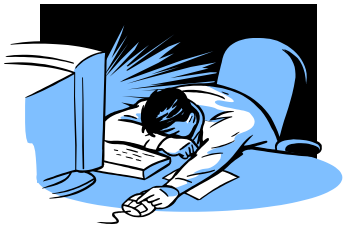
Requirements for specification & modeling techniques: 3. Timing (2)

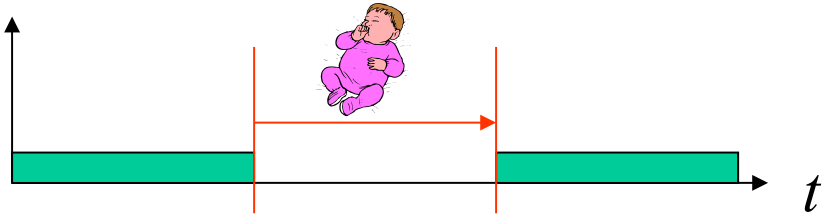
4 types of timing specs required, according to Burns, 1990:

1. Measure elapsed time
   Check, how much time has elapsed since last call



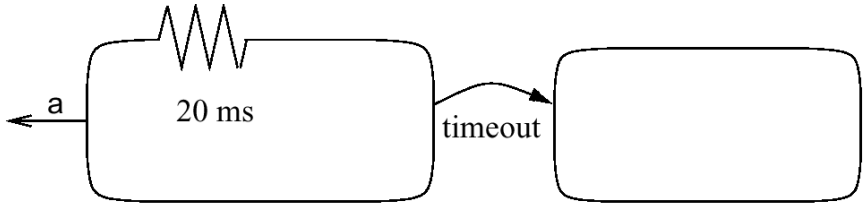
2. Means for delaying processes



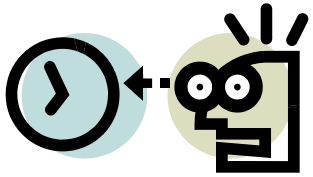


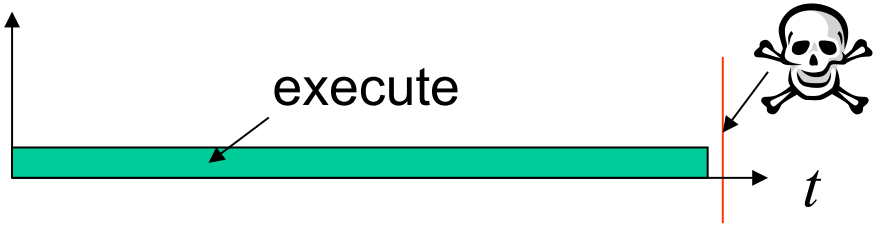
Requirements for specification & modeling techniques: 3. Timing (3)

3. Possibility to specify timeouts
   Stay in a certain state a maximum time.



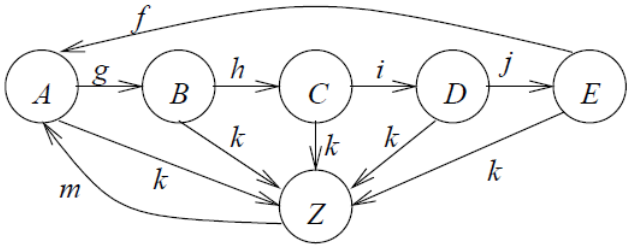
4. Methods for specifying deadlines
   Not available or in separate control file.





Requirements for specification of ES: 4. Support for designing reactive systems

- **State-oriented behavior**
  Required for reactive systems; classical automata insufficient.
- **Event-handling**
  (external or internal events)
- **Exception-oriented behavior**
  Not acceptable to describe exceptions for every state



We will see, how all the arrows labeled k can be replaced by a single one.

Requirements for specification & modeling techniques: 5...

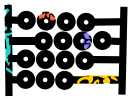
- **Presence of programming elements**
- **Executability** (no algebraic specification)
- **Support for the design of large systems** (☞ OO)
- **Domain-specific support**
- **Readability**
- **Portability and flexibility**
- **Termination**
- **Support for non-standard I/O devices**
- **Non-functional properties**
- **Support for the design of dependable systems**
- **No obstacles for efficient implementation**
- **Adequate model of computation**

  What does it mean “to compute”?









Problems with classical CS theory and von Neumann (thread) computing

Even the core … notion of "computable" is at odds with the requirements of embedded software.

In this notion, useful computation terminates, but termination is undecidable.

In embedded software, termination is failure, and yet to get predictable timing, subcomputations must decidably terminate.

What is needed is nearly a reinvention of computer science.

Edward A. Lee: Absolutely Positively on Time, *IEEE Computer*, July, 2005

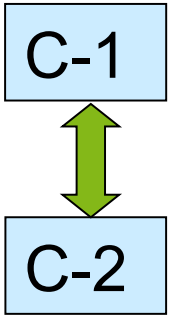
☞ Search for non-thread-based, non-von-Neumann MoCs.

Models of computation

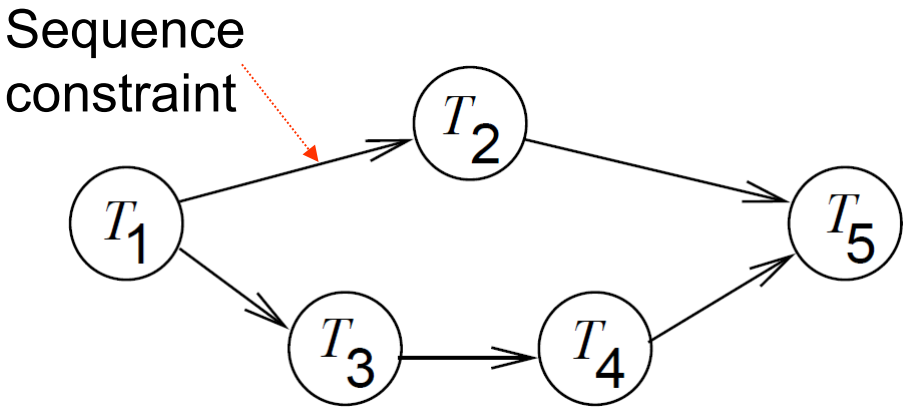
What does it mean, "to compute"?

Models of computation define:

- Components and an execution model for computations for each component
- Communication model for exchange of information between components.



Dependence graph: Definition



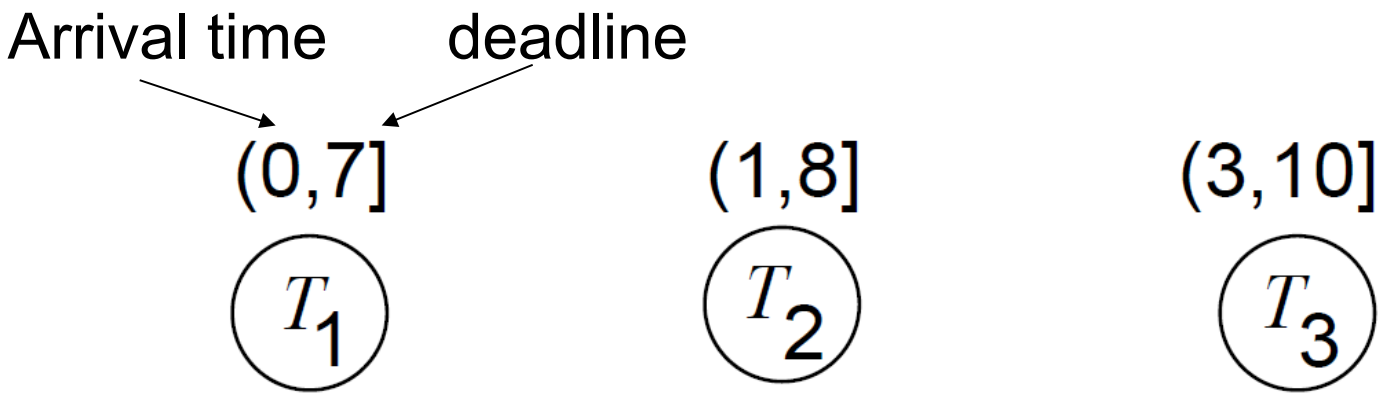
Nodes could be programs or simple operations

Def.: A **dependence graph** is a directed graph $G=(V,E)$ in which $E \subseteq V \times V$ is a relation.
If $(v_1, v_2) \in E$, then v_1 is called an **immediate predecessor** of v_2 and v_2 is called an **immediate successor** of v_1.
Suppose E^* is the transitive closure of E.
If $(v_1, v_2) \in E^*$, then v_1 is called a **predecessor** of v_2 and v_2 is called a **successor** of v_1.

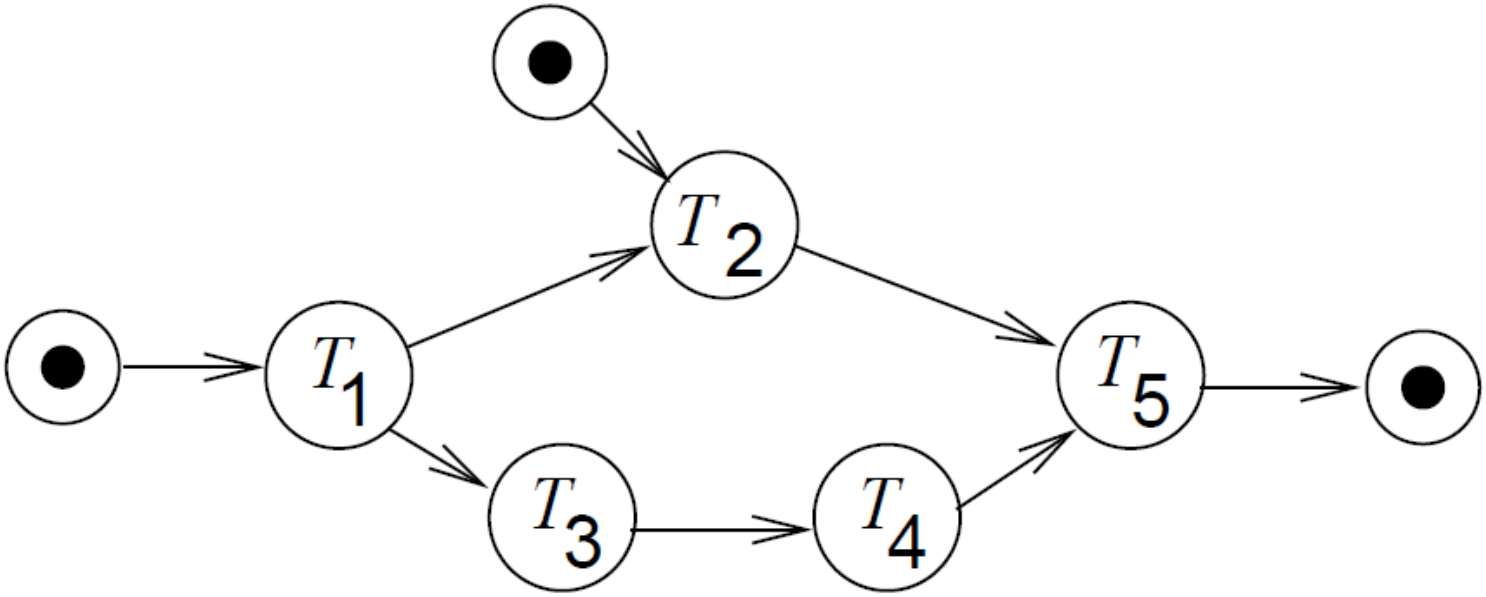
Dependence graph: Timing information

Dependence graphs may contain additional information, for example:

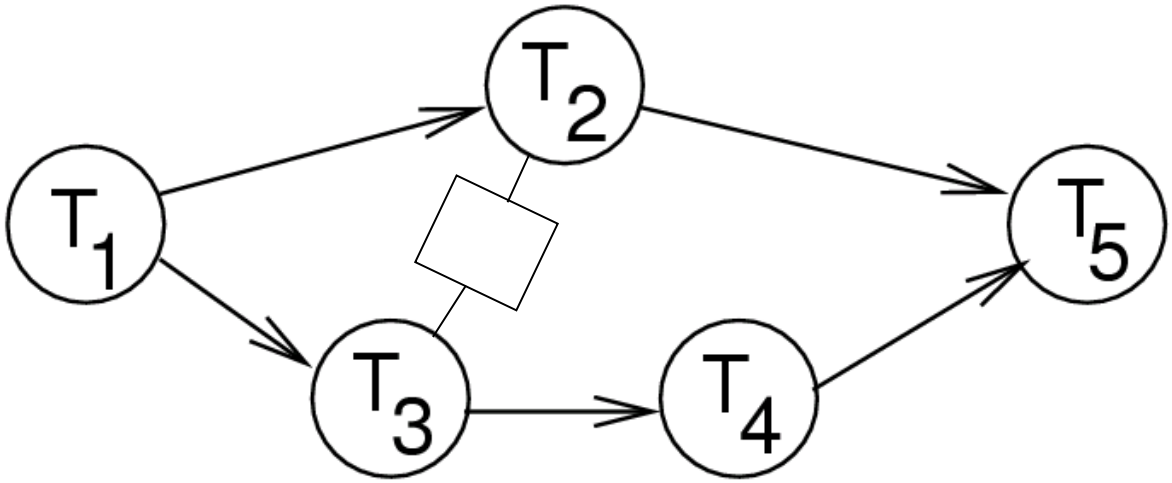
- Timing information

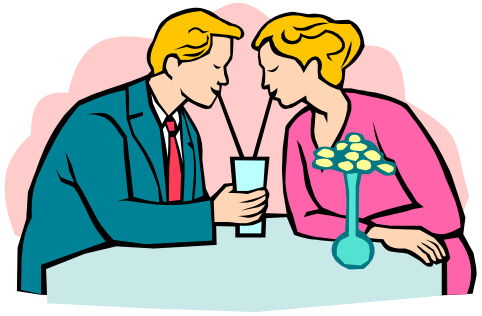


Dependence graph: I/O-information

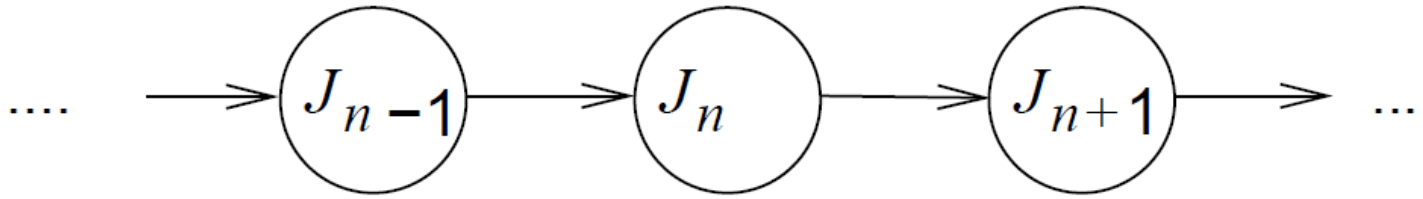


Dependence graph: Shared resources



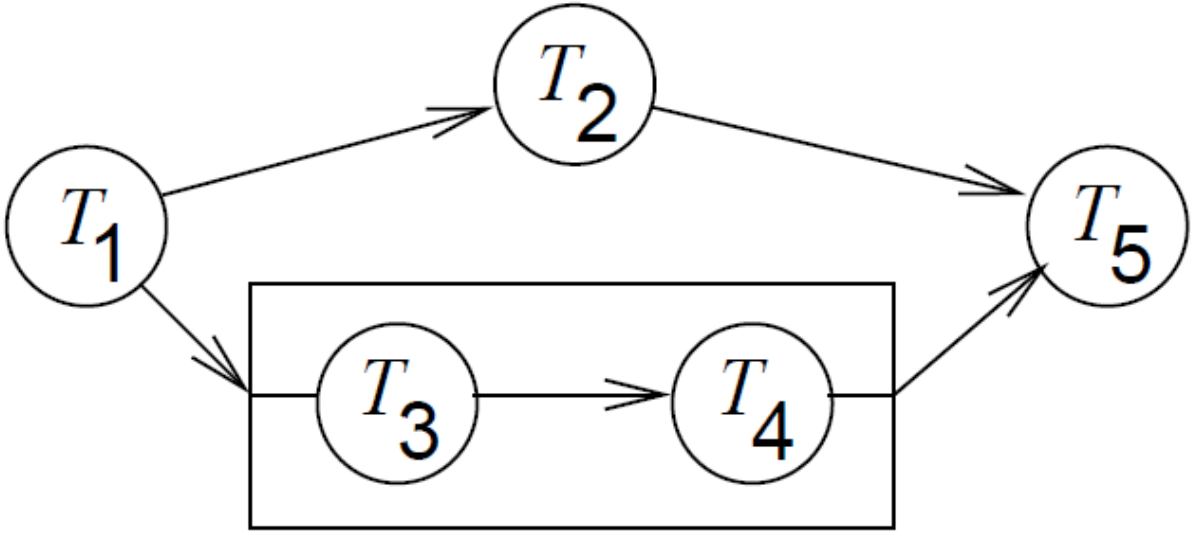


Dependence graph: Periodic schedules



- A **job** is single execution of the dependence graph
- Periodic dependence graphs are infinite

Dependence graph: Hierarchical task graphs



Communication

- **Shared memory**

Variables accessible to several components/tasks.

Model mostly restricted to local systems.

Shared memory

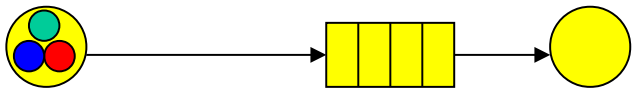
```
thread a {
  u = 1; ..
  P(S)  //obtain mutex
  if u<5 {u = u + 1; ..}
  // critical section
  V(S)  //release mutex
}
```

```
thread b {
  ..
  P(S)  //obtain mutex
  u = 5
  // critical section
  V(S)  //release mutex
}
```

- Unexpected u=6 possible if P(S) and V(S) is not used (double context switch before execution of {u = u+1}
- S: semaphore
- P(S) grants up to n concurrent accesses to resource
- n=1 in this case (mutex/lock)
- V(S) increases number of allowed accesses to resource
- Thread-based (imperative) model should be supported by mutual exclusion for critical sections

Non-blocking/asynchronous message passing

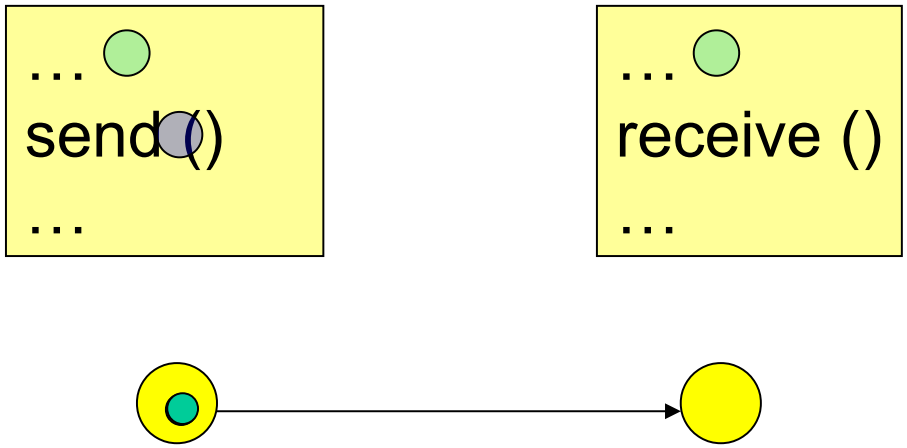
Sender does not have to wait until message has arrived



Potential problem: buffer overflow

Blocking/synchronous message passing - *rendez-vous*

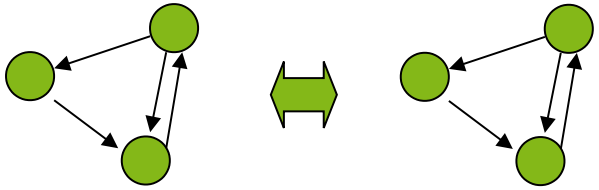
Sender will wait until receiver has received message



No buffer overflow, but reduced performance.

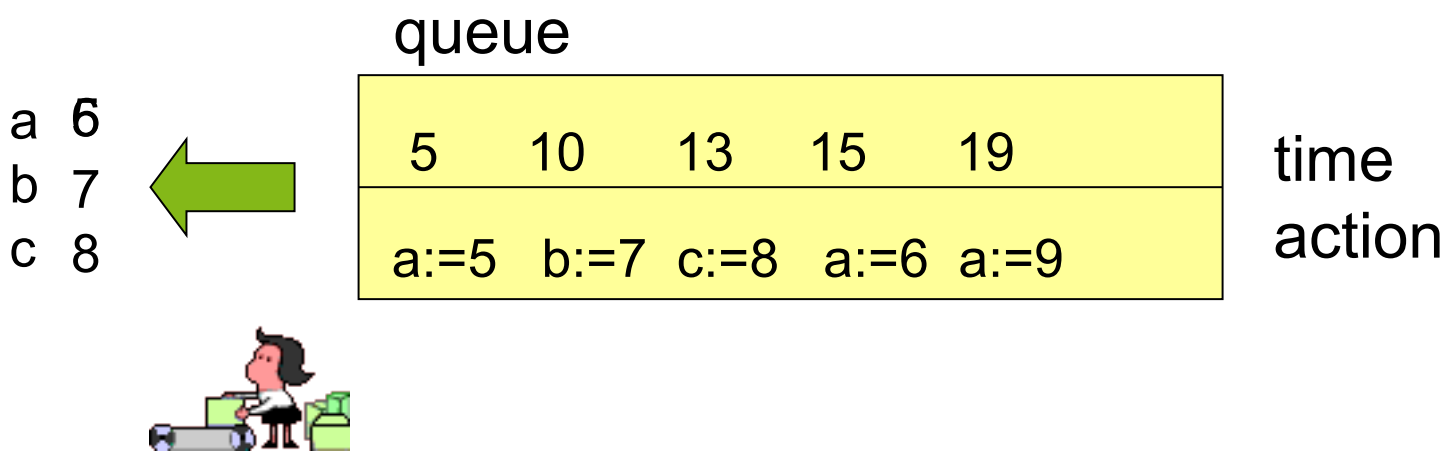
Organization of computations within the components (1)

- Finite state machines



Organization of computations within the components (2)

- Discrete event model



- Von Neumann model

  Sequential execution, program memory etc.

Organization of computations within the components (3)

- Differential equations

$$\frac{\partial^2 x}{\partial t^2} = b$$

- Data flow
  (models the flow of data in a distributed system)

- Petri nets
  (models synchronization in a distributed system)

Models of computation considered in this course

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorith. ☞ Computer arch.		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC* …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

Summary

Requirements for specification & modeling

- Hierarchy
- ...
- Appropriate model of computation

Models of computation =

- Dependence graphs
- models for communication
  - Shared memory
  - Message passing
- models of components
  - finite state machines (FSMs)
  - discrete event systems, …

Models of computation

Peter Marwedel
TU Dortmund,
Informatik 12

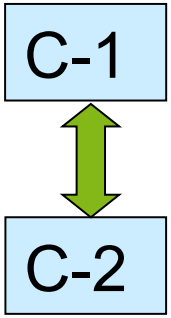
2012 年 10 月 23 日

Models of computation

What does it mean, "to compute"?

Models of computation define:

- Components and an execution model for computations for each component
- Communication model for exchange of information between components.



Models of computation considered in this course

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorith. ☞ Computer arch.		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC* …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

Why not use von-Neumann (thread-based) computing (C, C++, Java, …) ?

Potential race conditions (☞inconsistent results possible)

☞ Critical sections = sections at which exclusive access to resource r (e.g. shared memory) must be guaranteed.

```
thread a {
  ..
  P(S)  //obtain lock
  ..    // critical section
  V(S)  //release lock
}
```

```
thread b {
  ..
  P(S)  //obtain lock
  ..    // critical section
  V(S)  //release lock
}
```

Race-free access to shared memory protected by S possible

This model may be supported by:

- mutual exclusion for critical sections
- special memory properties

Why not just use von-Neumann computing (C, Java, …) (2)?

Problems with von-Neumann Computing

- Thread-based multiprocessing may access global variables
- We know from the theory of operating systems that
  - access to global variables might lead to race conditions,
  - to avoid these, we need to use mutual exclusion,
  - mutual exclusion may lead to deadlocks,
  - avoiding deadlocks is possible only if we accept performance penalties.
- Other problems (need to specify total orders, …)

Consider a Simple Example

"*The Observer pattern defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically*."

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*, Addison-Wesley, 1995

Example: Observer Pattern in Java

```
public void addListener(listener) {…}

public void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
      myListeners[i].valueChanged(newvalue)
    }
}
```

Would this work in a multithreaded context?

Thanks to Mark S. Miller for the details of this example.

Example: Observer Pattern with Mutual Exclusion (mutexes)

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
      myListeners[i].valueChanged(newvalue)
    }
}
```

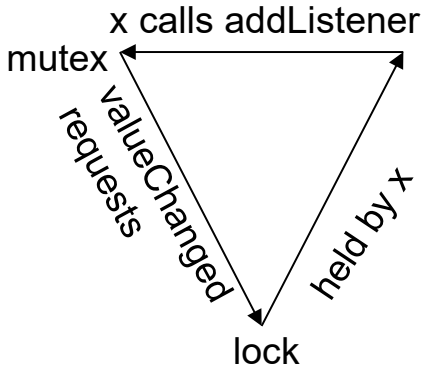
Javasoft recommends against this.

What's wrong with it?

Mutexes using monitors are minefields

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
      myListeners[i].valueChanged(newvalue)
    }
}
```



valueChanged() may attempt to acquire a lock on some other object and stall. If the holder of that lock calls addListener(): deadlock!

Simple Observer Pattern
Becomes not so simple

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized (this) {
        myValue=newValue;
        listeners=myListeners.clone();
    }
    for (int i=0; i<listeners.length; i++) {
      listeners[i].valueChanged(newValue)
    }
}
```

while holding lock, make a copy of listeners to avoid race conditions

notify each listener outside of the synchronized block to avoid deadlock

This still isn't right.
What's wrong with it?

Simple Observer Pattern
How to Make it Right?

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized (this) {
        myValue=newValue;
        listeners=myListeners.clone();
    }
    for (int i=0; i<listeners.length; i++) {
      listeners[i].valueChanged(newValue)
    }
}
```

Suppose two threads call setValue(). One of them will set the value last, leaving that value in the object, but listeners may be notified in the opposite order. The listeners may be alerted to the value-changes in the wrong order!

Why are deadlocks possible?

We know from the theory of operating systems, that deadlocks are possible in a multi-threaded system if we have

- Mutual exclusion
- Holding resources while waiting for more
- No preemption
- Circular wait

Conditions are met for our example

A stake in the ground …

Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans.

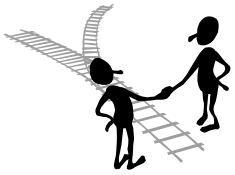
*"… **threads as a concurrency model are a poor match for embedded systems**. … they work well only … where best-effort scheduling policies are sufficient."*

Edward Lee: Absolutely Positively on Time, *IEEE Computer*, July, 2005

Ways out of this problem

- Looking for other options (“model-based design”)
- No model that meets all modeling requirements

☞ using compromises



Models of computation considered in this course

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorith. ☞ Computer arch.		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC* …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

Early design phase

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 10 月 23 日

Capturing the requirements as text

- In the very early phases of some design project, only descriptions of the system under design (SUD) in a natural language such as English or Japanese exist.
- Expectations for tools:
  - Machine-readable
  - Version management
  - Dependency analysis
  - Example: DOORS® [Telelogic/IBM]

Use cases

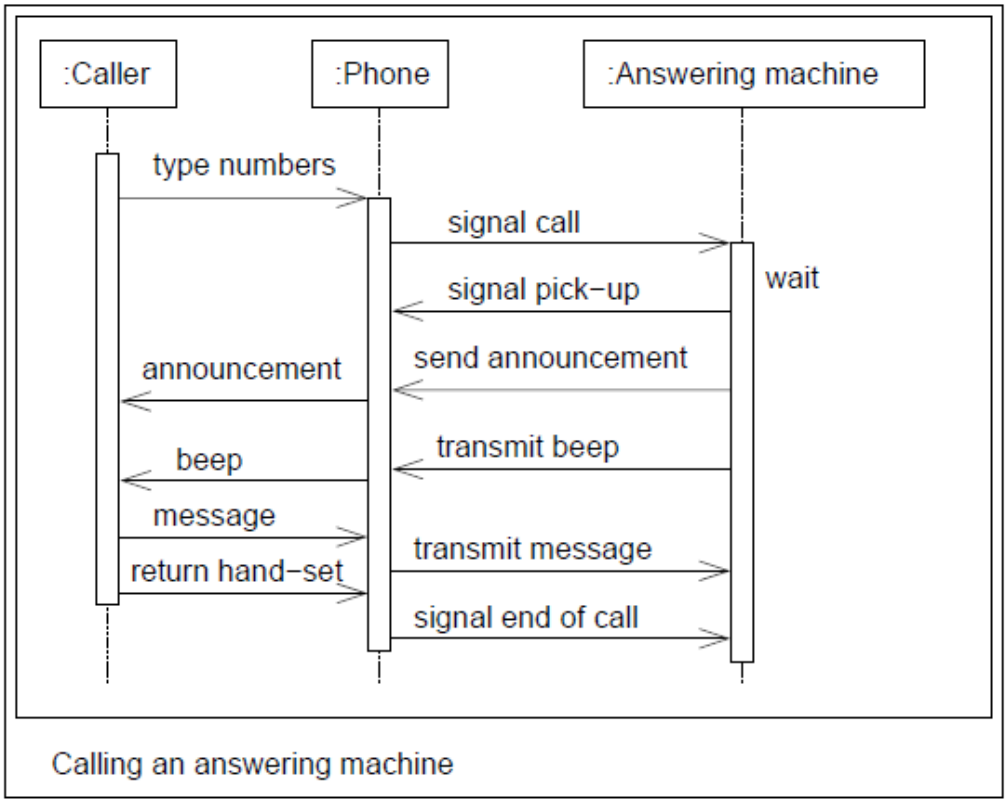
- Use cases describe possible applications of the SUD
- Included in UML (Unified Modeling Language)
- Example: Answering machine

- Neither a precisely specified model of the computations nor a precisely specified model of the communication

(Message) Sequence charts

- Explicitly indicate exchange of information
- One dimension (usually vertical dimension) reflects time
- The other reflects distribution in space

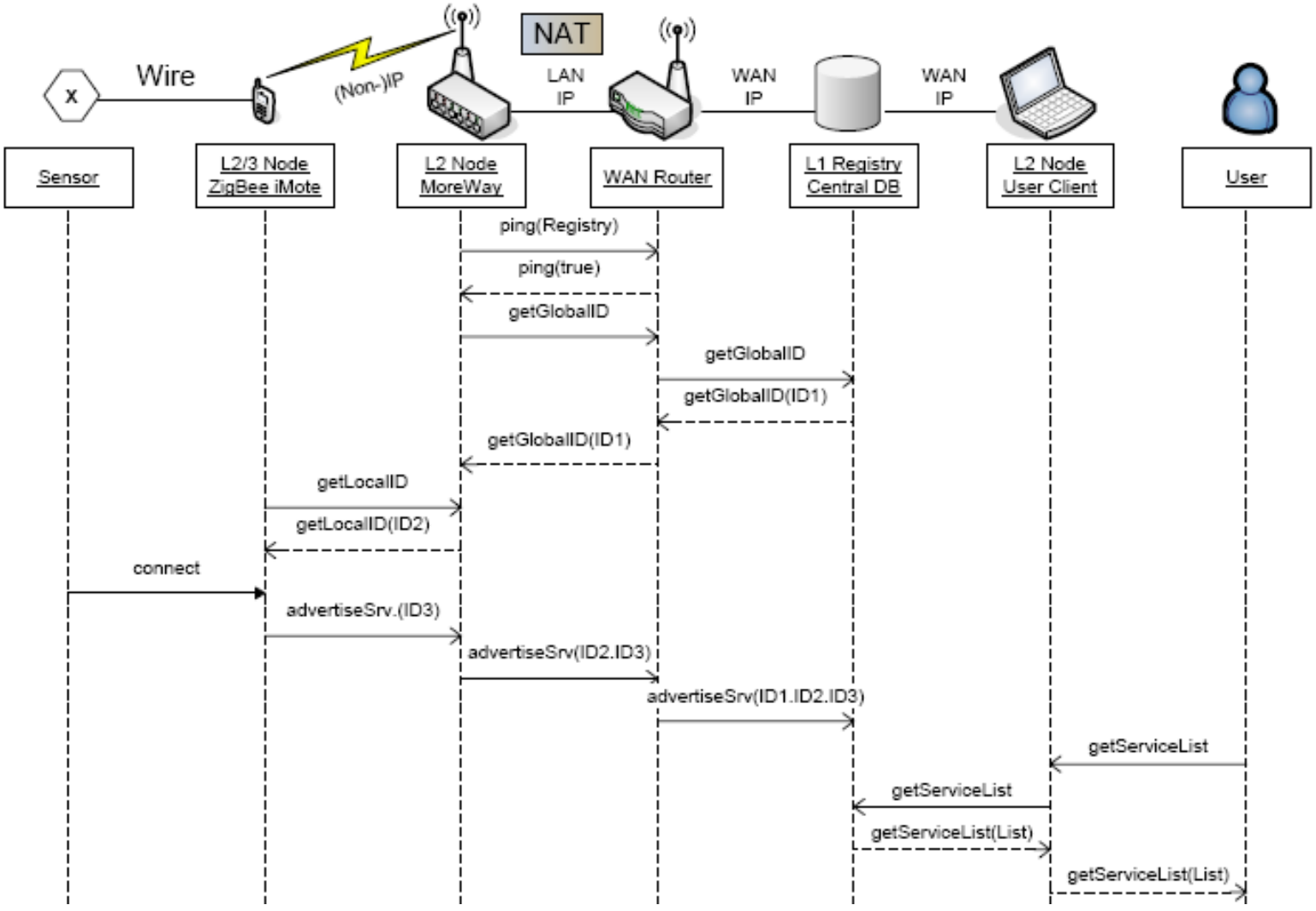
Example:



Calling an answering machine

- Included in UML
- Earlier called *Message Sequence Charts*
  now mostly called *Sequence Charts*

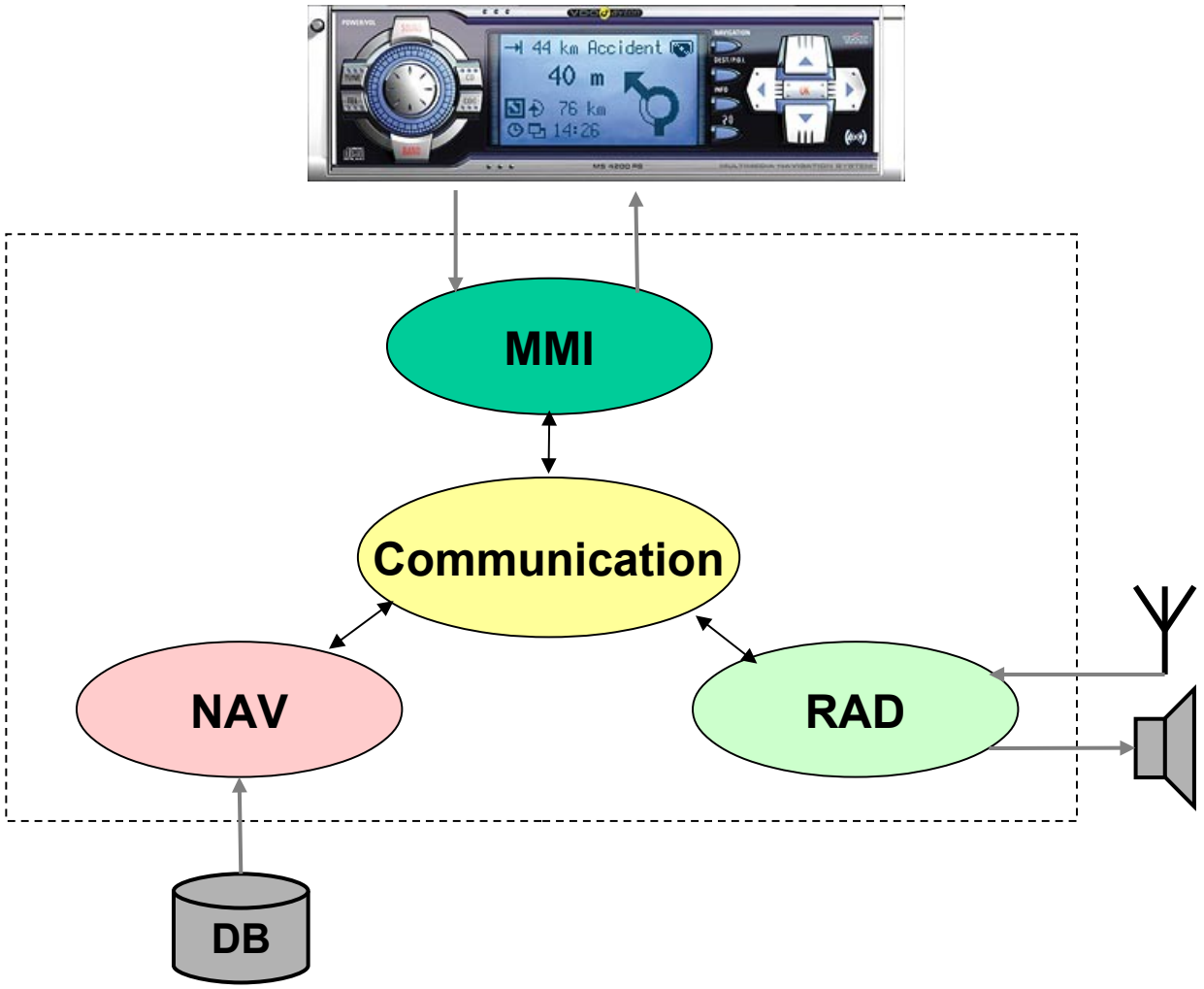
Example (2)


Wire
(Non-)IP
NAT
LAN IP
WAN IP
WAN IP
Sensor
L2/3 Node ZigBee iMote
L2 Node MoreWay
WAN Router
L1 Registry Central DB
L2 Node User Client
User
ping(Registry)
ping(true)
getGlobalID
getGlobalID
getGlobalID(ID1)
getGlobalID(ID1)
getLocalID
getLocalID(ID2)
connect
advertiseSrv.(ID3)
advertiseSrv(ID2.ID3)
advertiseSrv(ID1.ID2.ID3)
getServiceList
getServiceList
getServiceList(List)
getServiceList(List)

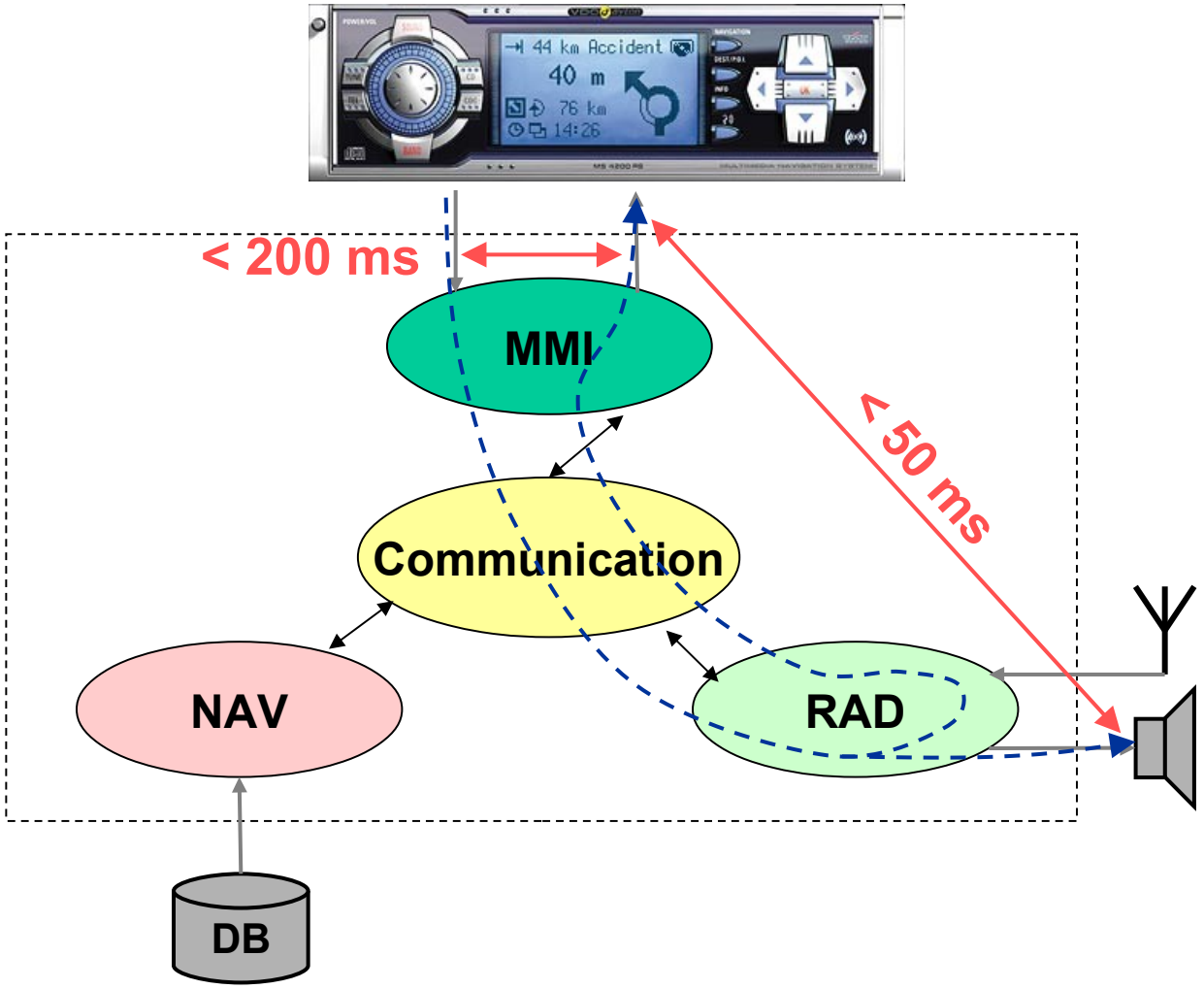
Application: In-Car Navigation System

- Car radio with navigation system
- User interface needs to be responsive
- Traffic messages (TMC) must be processed in a timely way
- Several applications may execute concurrently

System Overview

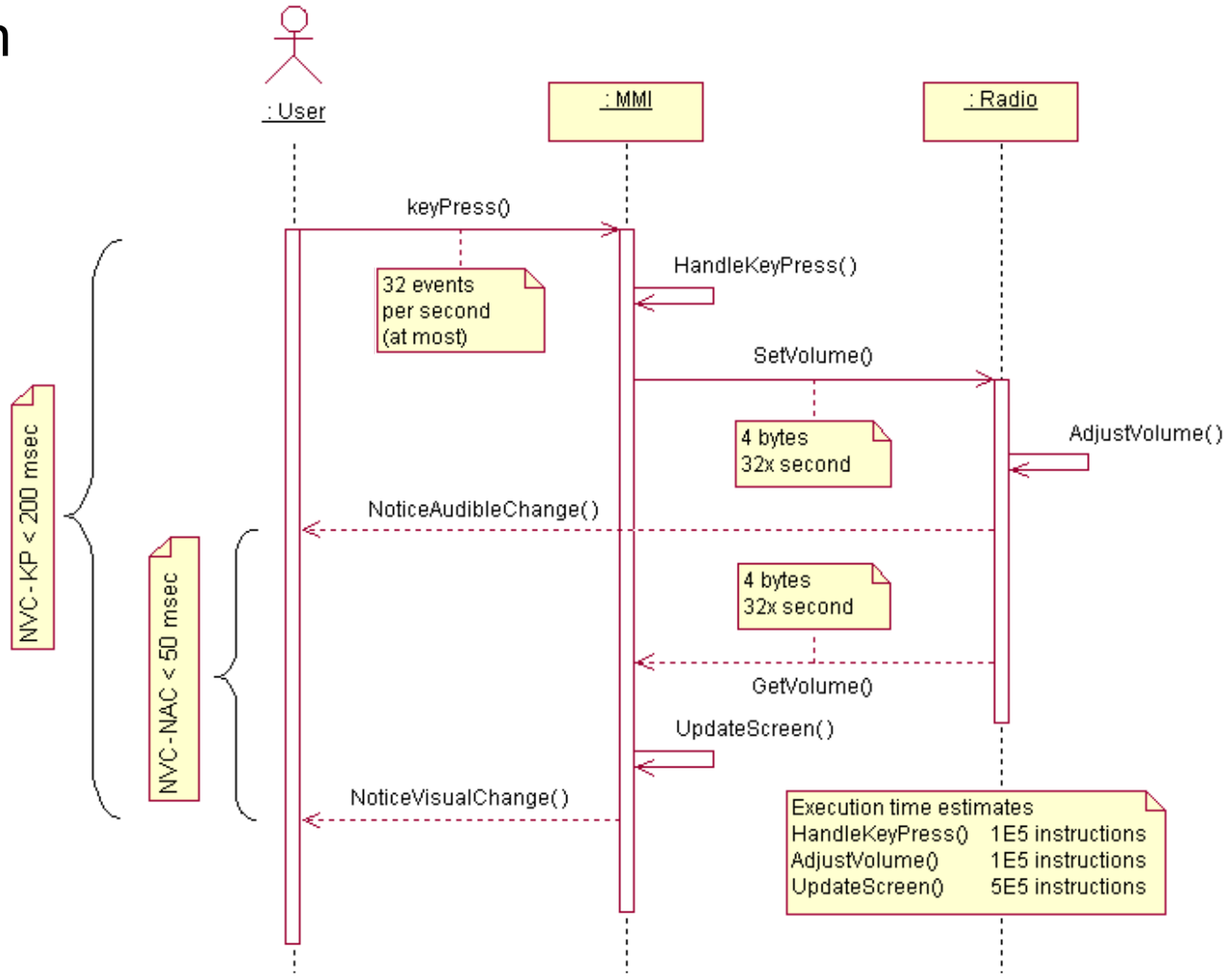


Use case 1: Change Audio Volume (1)

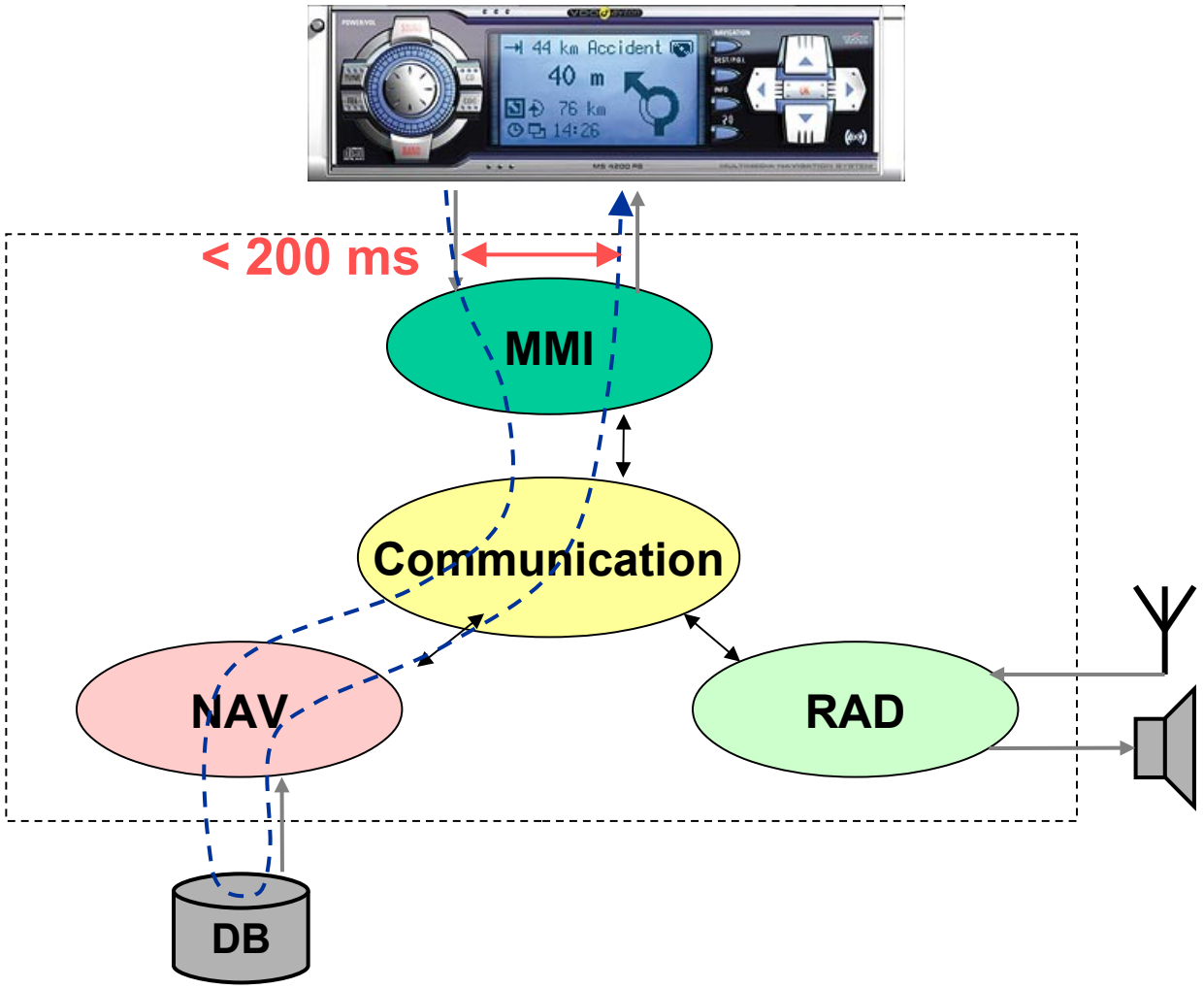


Use case 1: Change Audio Volume (2)

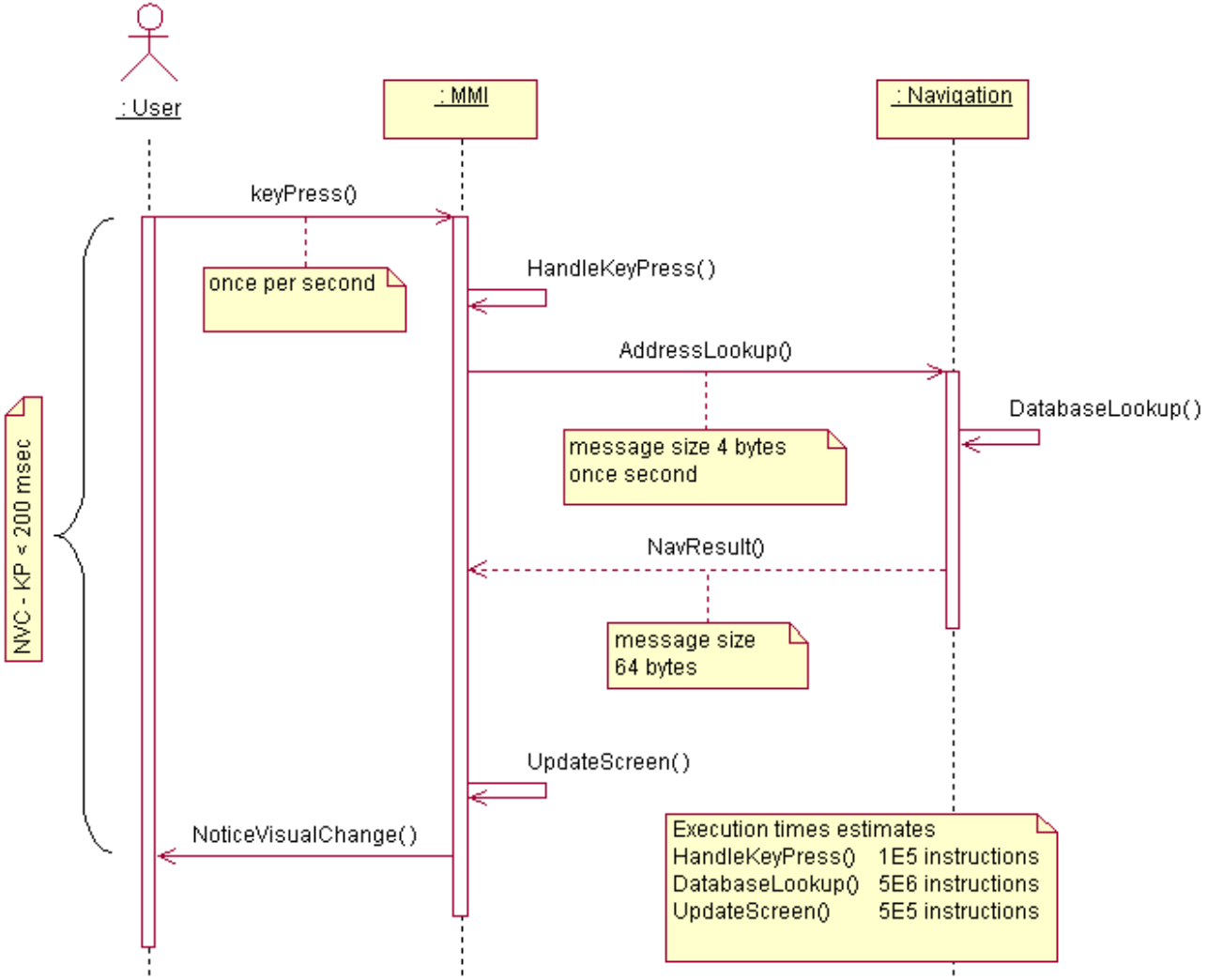
Communication
Resource
Demand



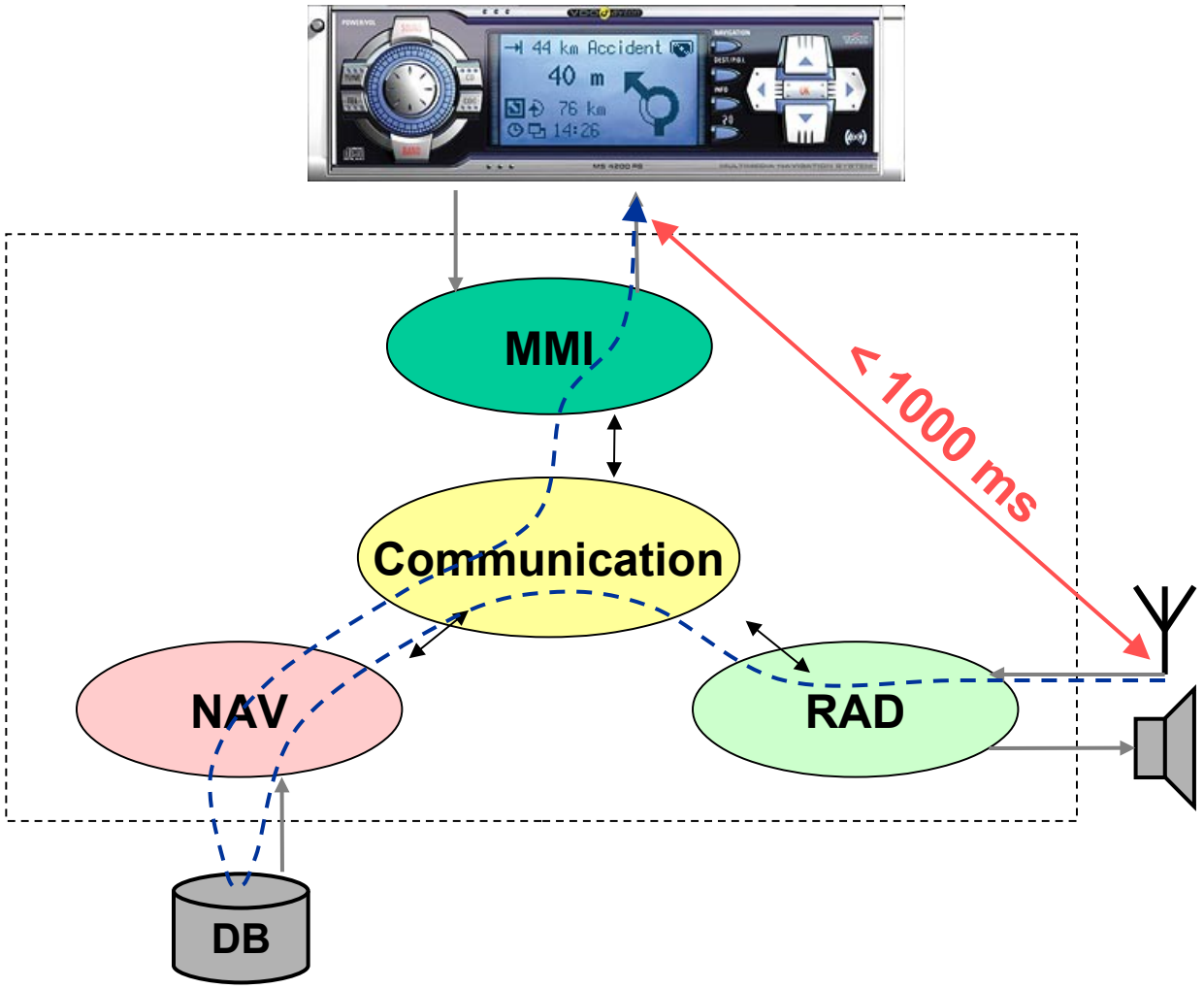
Use case 2: Lookup Destination Address (1)



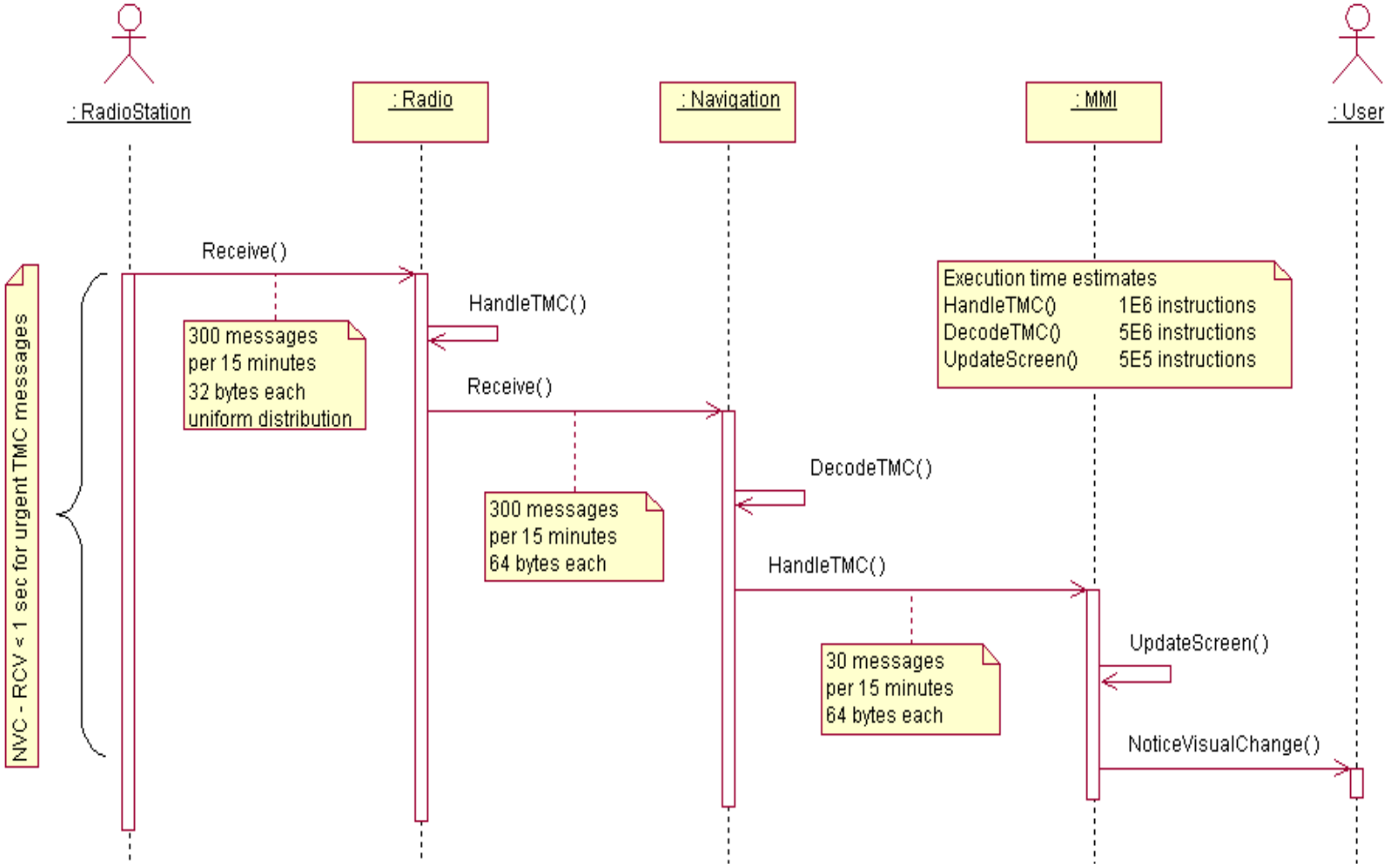
Use case 2: Lookup Destination Address (2)



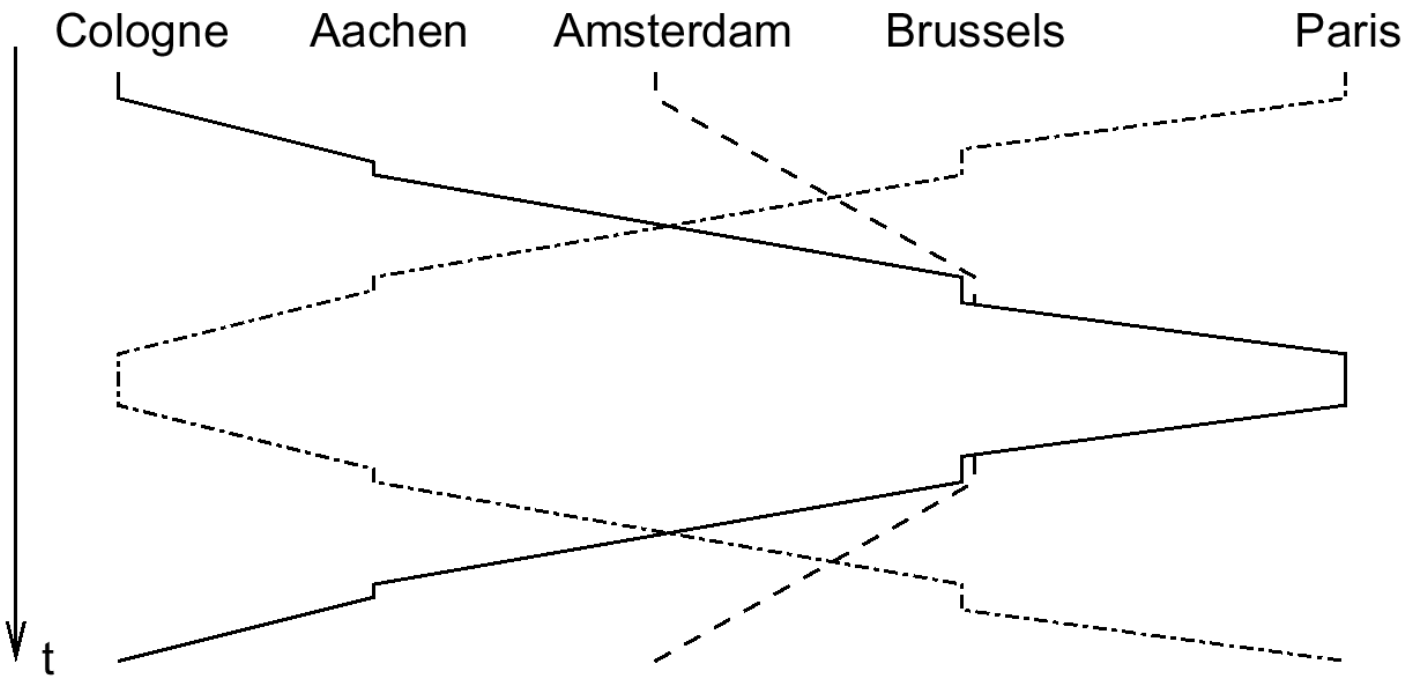
Use case 3: Receive TMC Messages (1)



Use case 3: Receive TMC Messages (2)

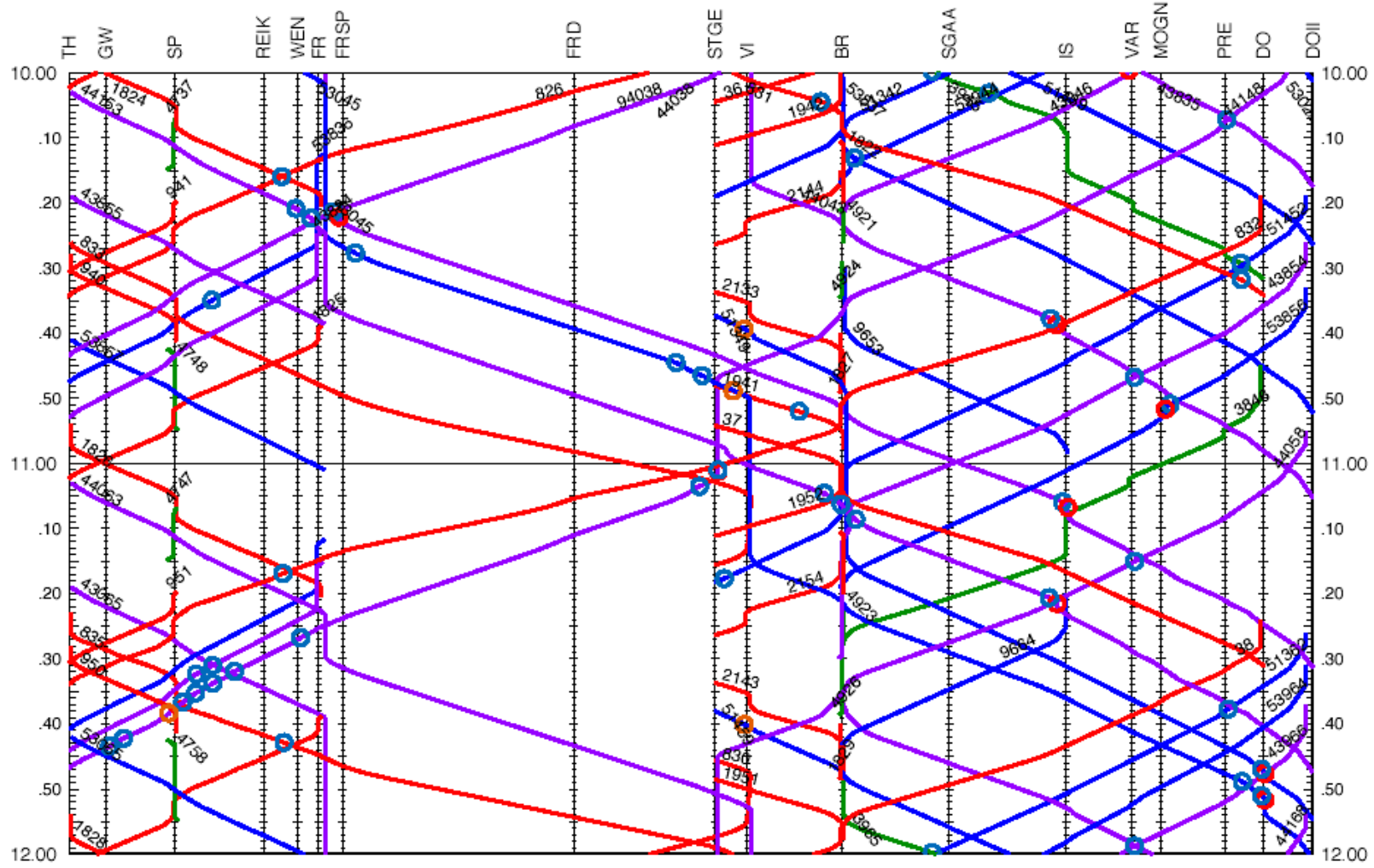


(Message) Sequence Charts (MSC)



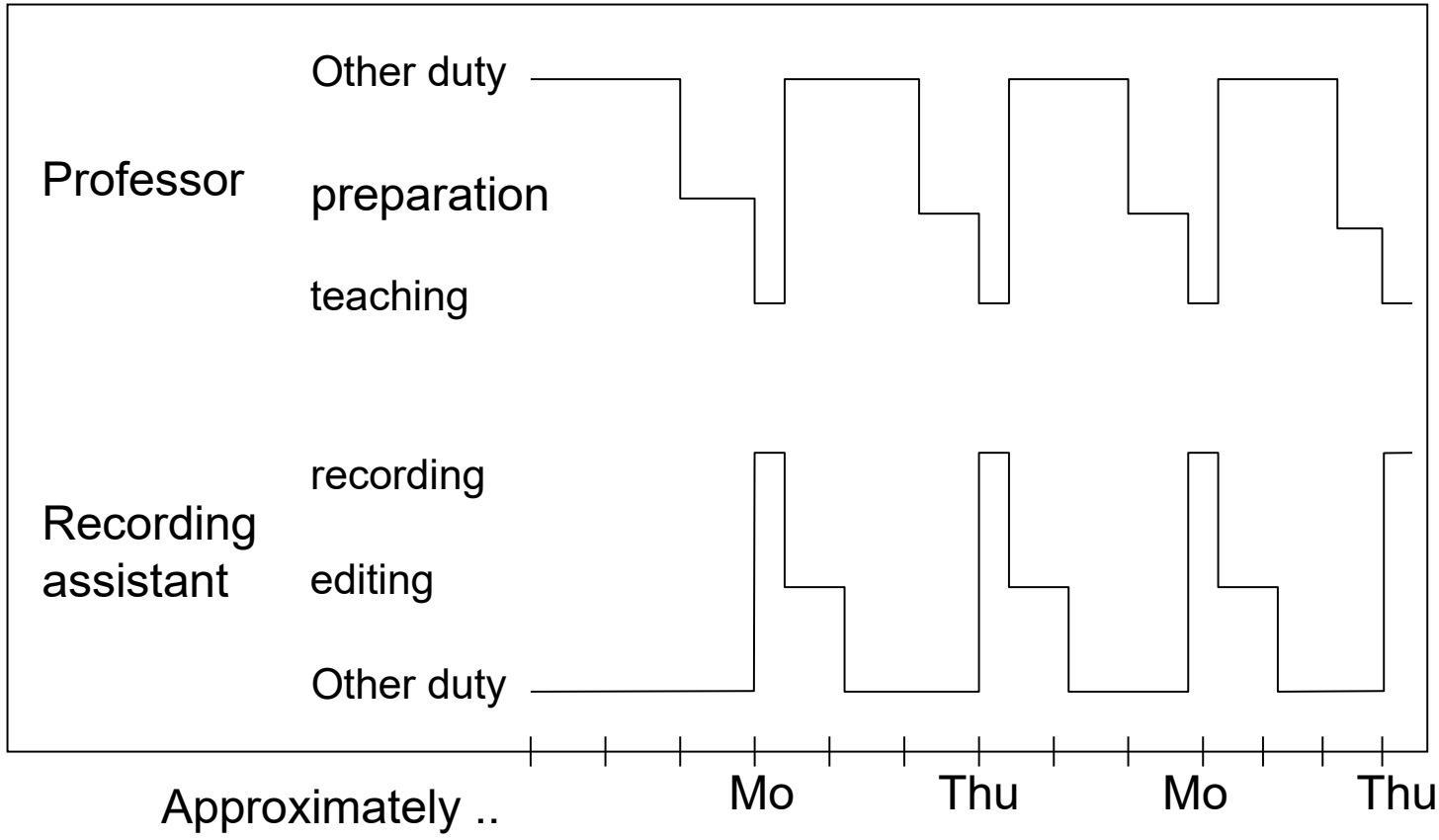
No distinction between accidental overlap and synchronization

Time/distance diagrams as a special case



UML: Timing diagrams

Can be used to show the change of the state of an object over time.



Based on Scott Ambler, Agile Modeling, 2003 www.agilemodeling.com

Life Sequence Charts* (LSCs)

Key problems observed with standard MSCs:

- During the design process, MSCs are initially interpreted as *"what could happen"*
  - ☞existential interpretation, still allowing other behaviors
- Later, they are frequently assumed to describe *„what must happen"*
  - ☞referring to what happens in the implementation

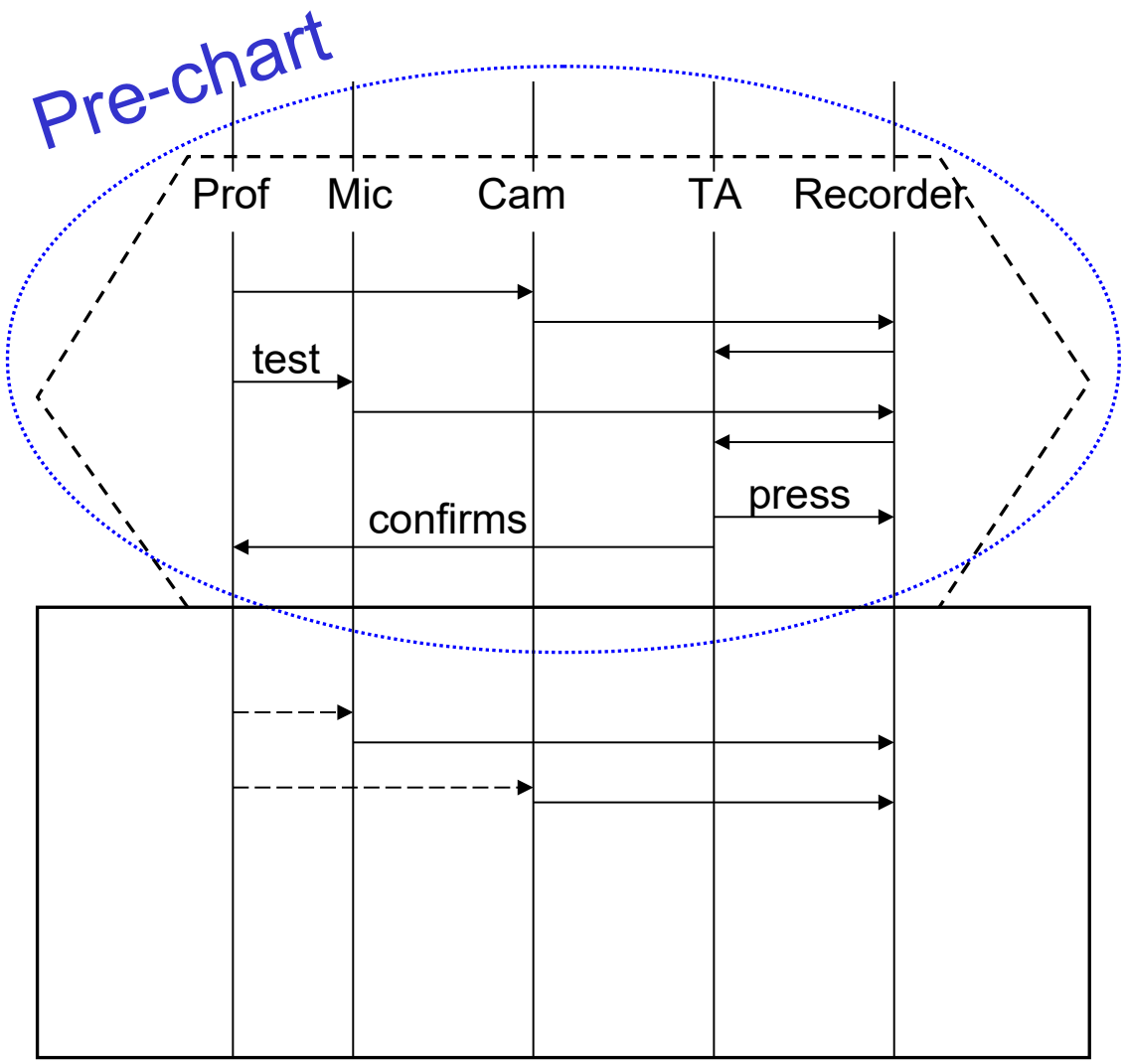
* W. Damm, D. Harel: LSCs: Breathing Life into Message Sequence Charts, *Formal Methods in System Design*, 19, 45–80, 2001

Extensions for LSCs

Extension 1:

Introduction of **pre-charts:**
Pre-charts describe conditions that must hold for the main chart to apply.

Example:



Extensions (2)

Extension 2: Mandatory vs. provisional behavior

Level	**Mandatory (solid lines)**	**Provisional (dashed lines)**
Chart	All runs of the system satisfy the chart	At least one run of the system satisfies the chart
Location	Instance must move beyond location/time	Instance run need not move beyond loc./time
Message	If message is sent, it will be received	Receipt of message is not guaranteed
Condition	Condition must be met; otherwise abort	If condition is not met, exit subchart

(Message) Sequence Charts

PROs:

- Appropriate for visualizing schedules,
- Proven method for representing schedules in transportation.
- Standard defined: *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC*), ITU-TS, Geneva, 1996.
- Semantics also defined: *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)—Annex* B: *Algebraic Semantics of Message Sequence Charts*, ITU-TS, Geneva.

CONs:

- describes just one case, no timing tolerances: "What *does an MSC specification mean: does it describe all behaviors of a system, or does it describe a set of sample behaviors of a system?" **

* H. Ben-Abdallah and S. Leue, "Timing constraints in message sequence chart specifications," in *Proc. 10th International Conference on Formal Description Techniques FORTE/PSTV'97*, Chapman and Hall, 1997.

Communicating finite state machines

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 10 月 23 日

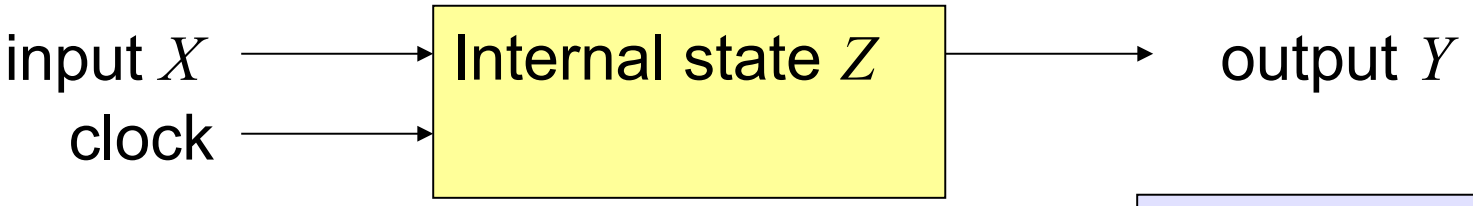
Models of computation considered in this course

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorith. ☞ Computer arch.		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC* …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

StateCharts: recap of classical automata

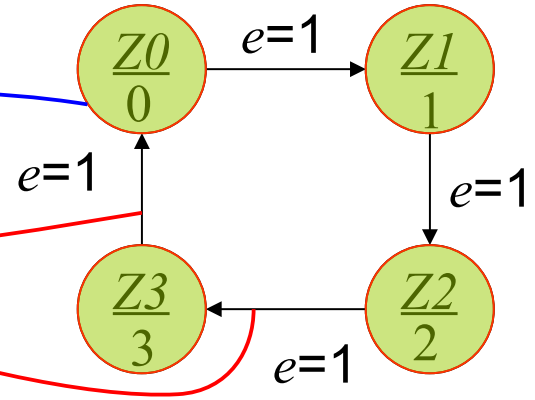
Classical automata:



Next state Z^+ computed by function δ
Output computed by function λ

Moore- + Mealy automata=finite state machines (FSMs)

- Moore-automata:
  $Y = \lambda\ (Z);\quad Z^+ = \delta\ (X, Z)$
- Mealy-automata
  $Y = \lambda\ (X, Z);\quad Z^+ = \delta\ (X, Z)$



FSM – **F**inite **S**tate **M**achine

- Deterministischer Endlicher Automat mit Ausgabe
- 2 äquivalente Modelle
  - Mealy: Ausgabe hängt *von Zustand und Eingabe* ab
  - Moore: –"– *nur vom Zustand* ab
- 6-Tupel $\langle Z, \Sigma, \Delta, \delta, \lambda, z_0 \rangle$
  - Z Menge von Zuständen
  - Σ Eingabealphabet
  - Δ Ausgabealphabet
  - δ Übergangsfunktion $\delta : Z \times \Sigma \rightarrow Z$
  - λ Ausgabefunktion $\lambda : Z \times \Sigma \rightarrow \Delta$ Mealy-Modell
    $\lambda : Z \rightarrow \Delta$ Moore- –"–
  - z_0 Startzustand

Mealy-Modell und Moore-Modell

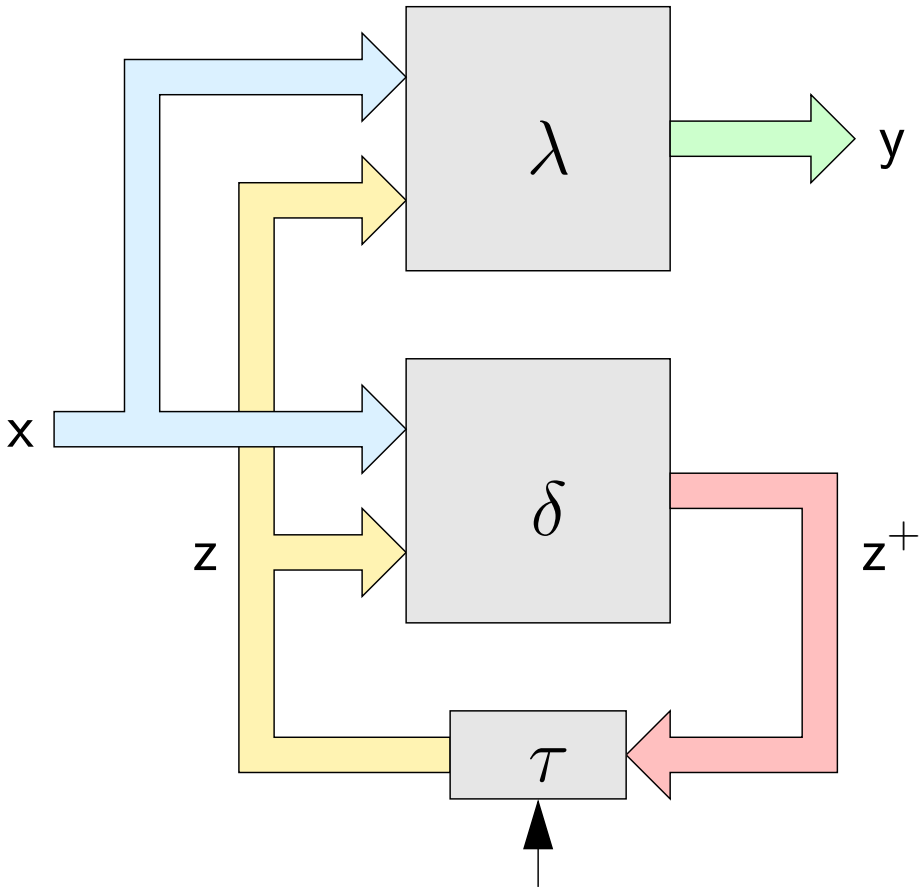
- **Mealy-Modell**: die Ausgabe hängt vom Zustand z und vom momentanen Input x ab
- **Moore-Modell**: die Ausgabe des Schaltwerks hängt nur vom aktuellen Zustand z ab

Ausgabefunktion:	$y = \lambda(z, x)$	Mealy
	$y = \lambda(z)$	Moore
Überführungsfunktion:	$z^+ = \delta(z, x)$	Moore und Mealy

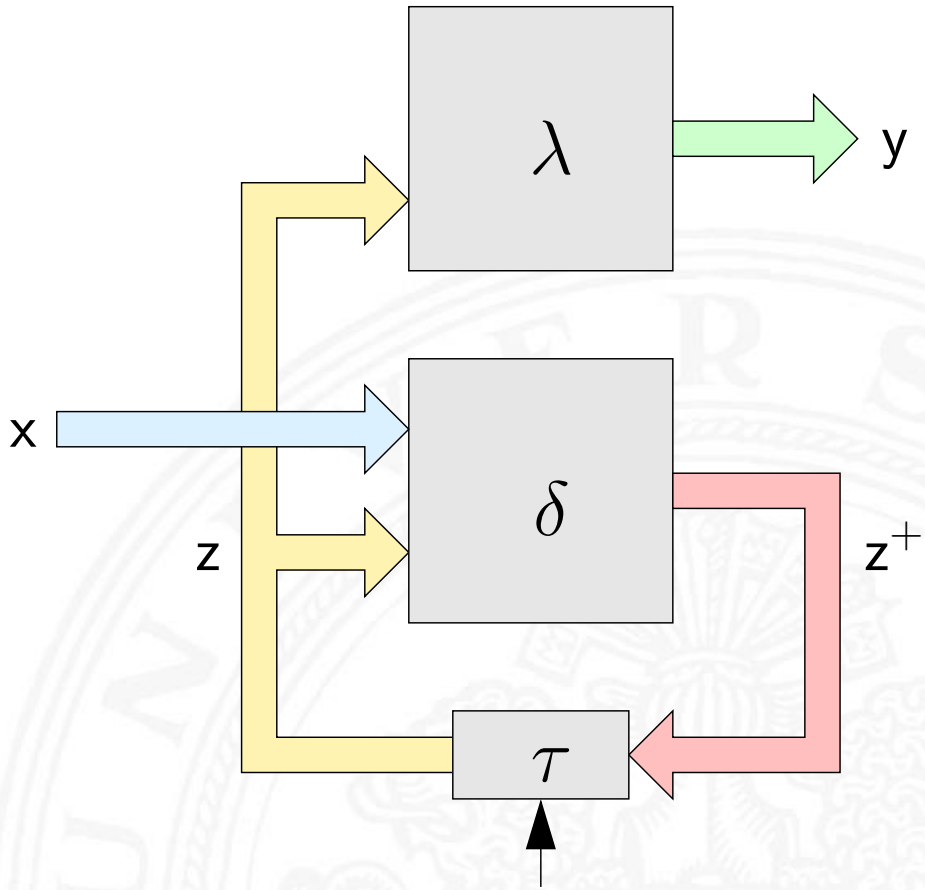
- **Speicherglieder** oder Verzögerung τ im Rückkopplungspfad

Mealy-Modell und Moore-Modell (cont.)

- Mealy-Automat



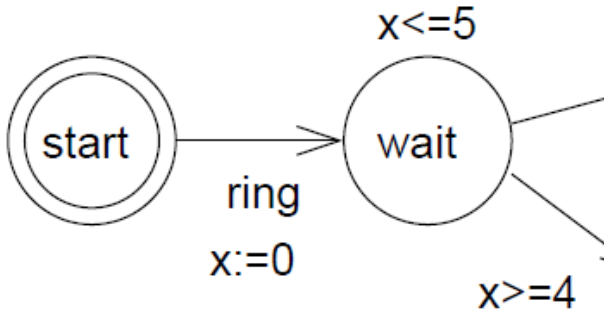
Moore-Automat



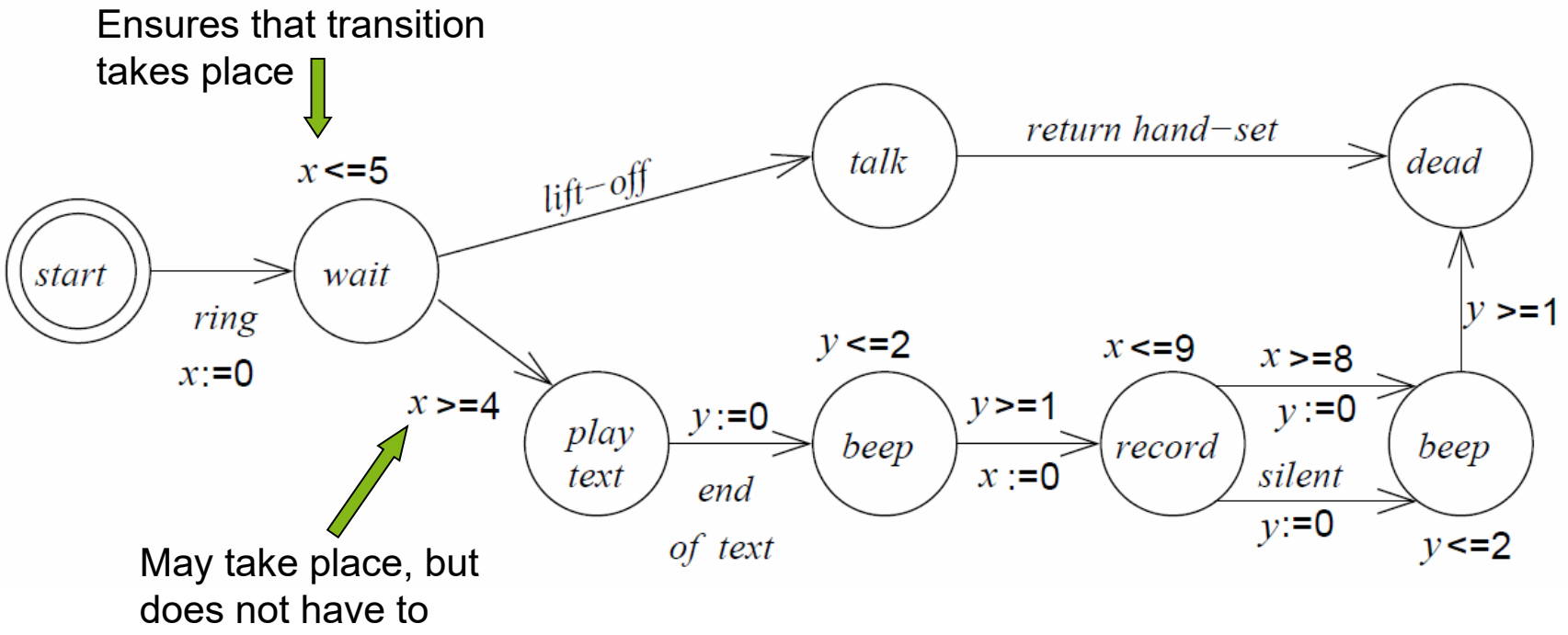
Timed automata

- Timed automata = automata + models of time
- The variables model the logical clocks in the system, that are initialized with zero when the system is started, and then increase synchronously with the same rate.
- Clock constraints i.e. guards on edges are used to restrict the behavior of the automaton.
  A transition represented by an edge can be taken when the clocks values satisfy the guard labeled on the edge.
- Additional invariants make sure, the transition is taken.
- Clocks may be reset to zero when a transition is taken.

[Bengtsson and Yi, 2004].



Example: Answering machine



Definitions

Let C: real-valued variables C representing clocks.

Let Σ: finite alphabet of possible inputs.

Definition: A **clock constraint** is a conjunctive formula of atomic constraints of the form
$x \circ n$ or $x - y \circ n$ for $x, y \in C$, $\circ \in \{\leq, <, =, >, \geq\}$ and $n \in N$

Let $B(C)$ be the set of clock constraints.

Definition: A **timed automaton** A is a tuple (S, s_0, E, I) where

S is a finite set of states, s_0 is the initial state,

$E \subseteq S \times B(C) \times \Sigma \times 2^C \times S$ is the set of edges,

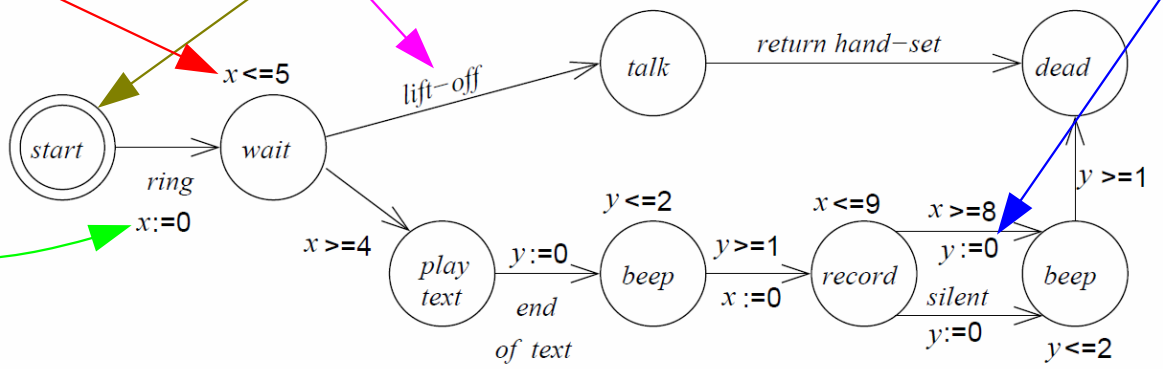
$B(C)$: conjunctive condition, 2^C: variables to be reset

$I : S \rightarrow B(C)$ is the set of invariants for each of the states

$B(C)$: invariant that must hold for state S

Definitions (2)

Let C: real-valued variables C representing clocks.
Let Σ: finite alphabet of possible inputs.
Definition: A **clock constraint** is a conjunctive formula of atomic constraints of the form $x \circ n$ or $x-y \circ n$ for $x, y \in C$, $\circ \in \{\leq,<,=,>,\geq\}$ and $n \in N$
Let $B(C)$ be the set of clock constraints.
Definition: A **timed automaton** A is a tuple (S, s_0, E, I) where S is a finite set of states, s_0 is the initial state,
$E \subseteq S \times B(C) \times \Sigma \times 2^C \times S$ is the set of edges, $B(C)$: conjunctive condition, 2^C: variables to be reset
$I : S \rightarrow B(C)$ is the set of invariants for each of the states, $B(C)$: invariant that must hold for state S



Summary

- Motivation for non-von Neumann models
- Support for early design phases
  - Text
  - Use cases
  - (Message) sequence charts
- Automata models
  - Timed automata

Communicating finite state machines

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 10 月 24 日

Models of computation considered in this course

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorith. ☞ Computer arch.		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC* …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

StateCharts

Classical automata not useful for complex systems (complex graphs cannot be understood by humans).

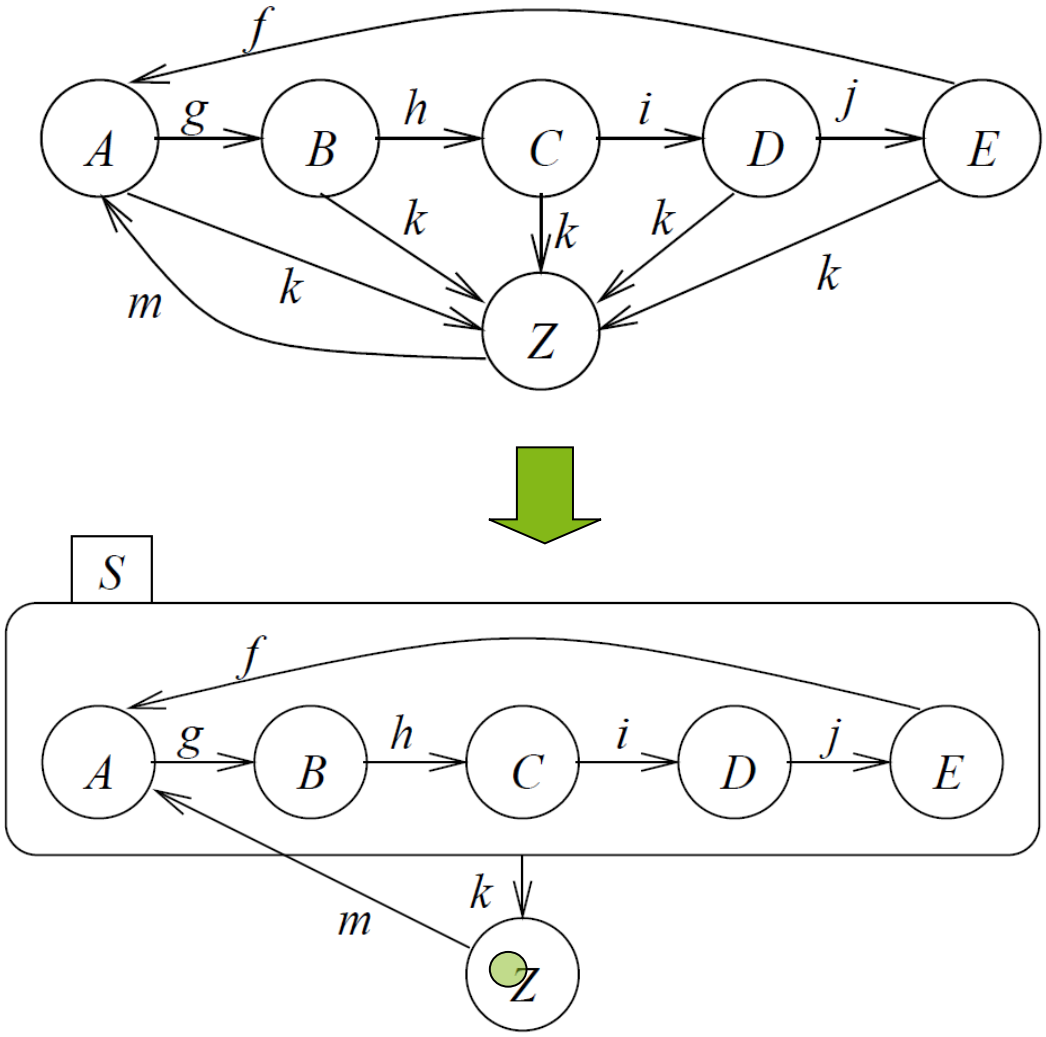
- Introduction of hierarchy
- StateCharts [Harel, 1987]

StateChart = *the only unused combination of „flow“ or „state“ with „diagram“ or „chart“*

Used here as a (prominent) example of a model of computation based on shared memory communication.

- appropriate only for local (non-distributed) systems

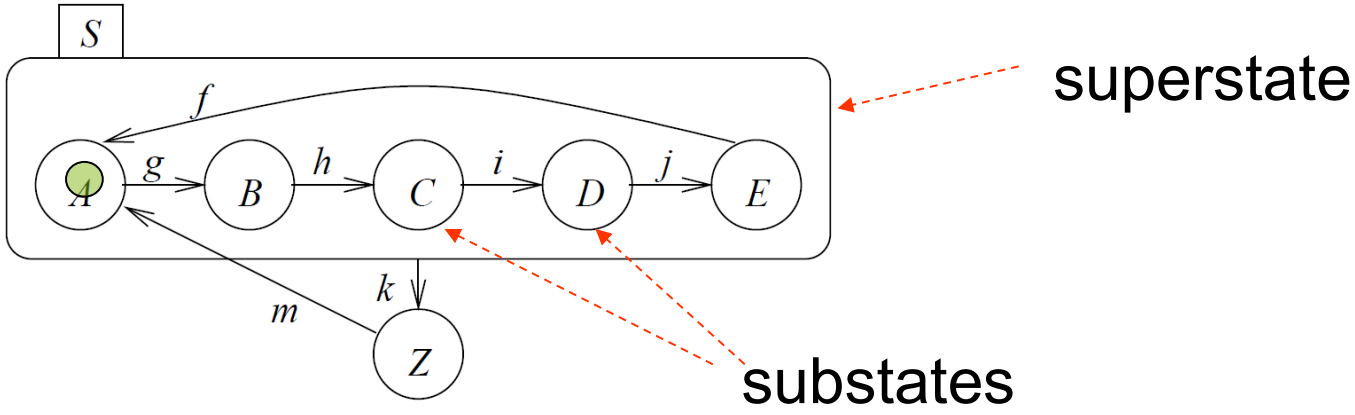
Introducing hierarchy



FSM will be **in** exactly one of the substates of S if S is **active** (either in A or in B or ..)

Definitions

- Current states of FSMs are also called **active** states.
- States which are not composed of other states are called **basic states.**
- States containing other states are called **super-states**.
- Super-states S are called **OR-super-states**, if exactly one of the sub-states of S is active whenever S is active.



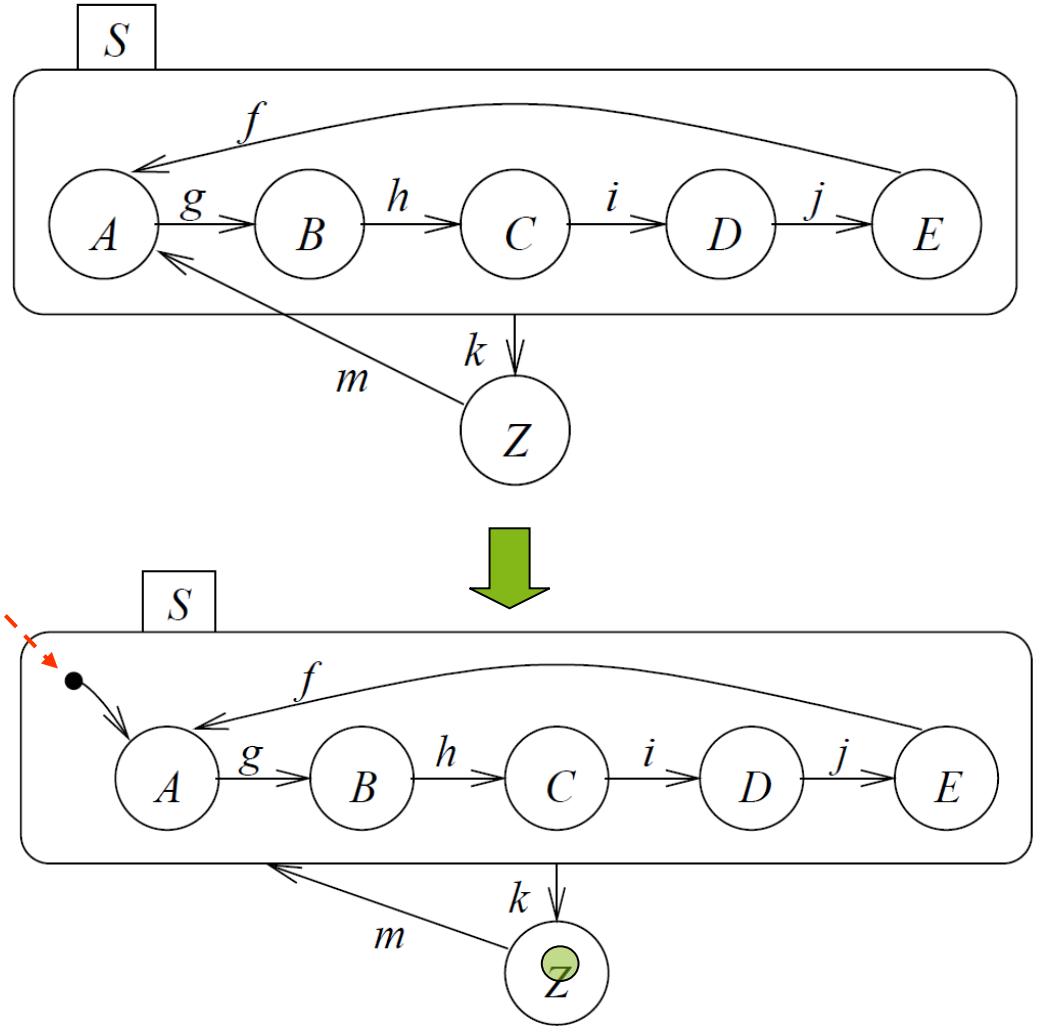
Default state mechanism

Try to hide internal structure from outside world!

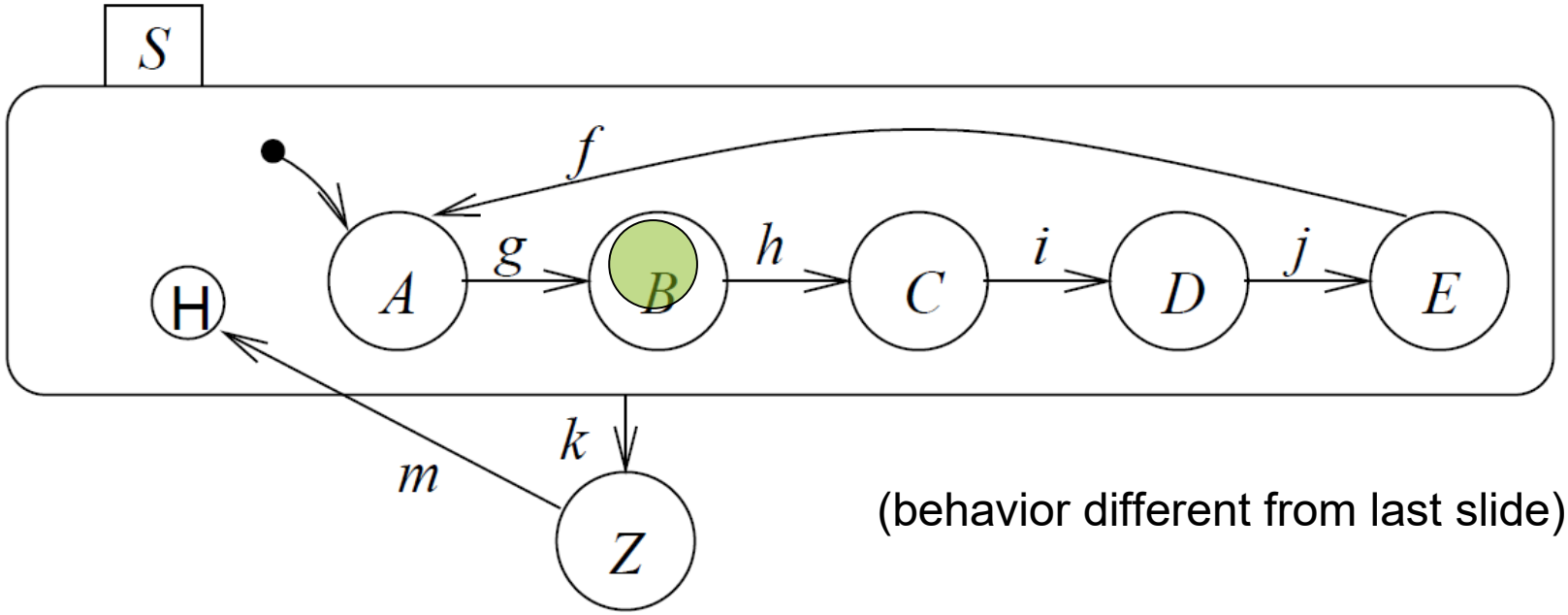
☞ Default state

Filled circle indicates sub-state entered whenever super-state is entered.

Not a state by itself!



History mechanism

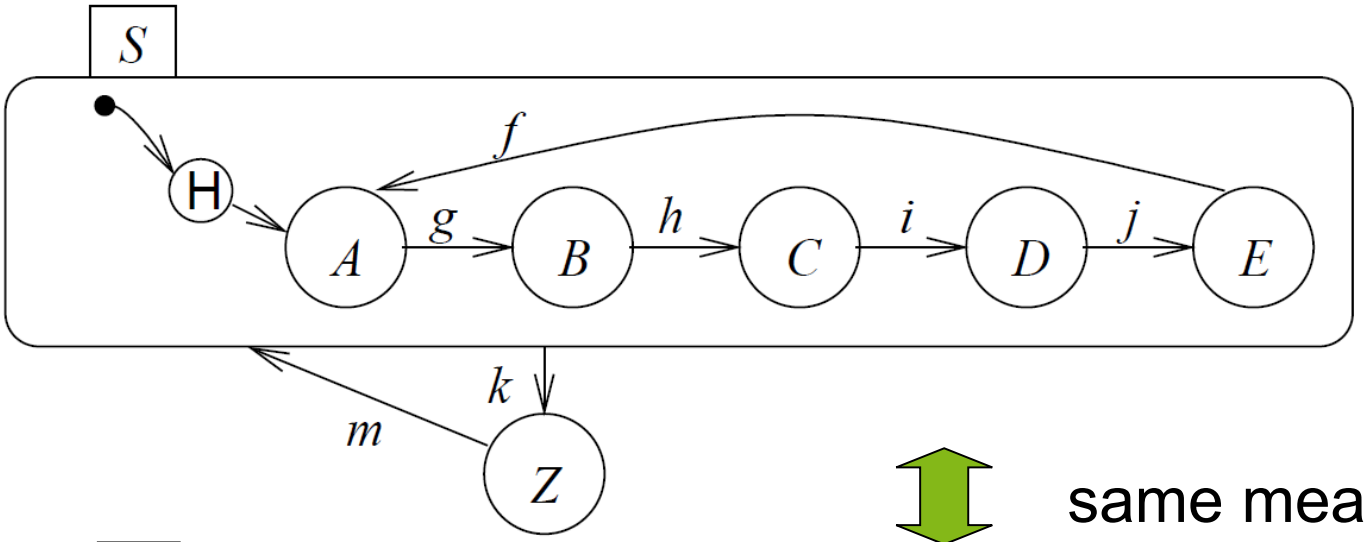


(behavior different from last slide)

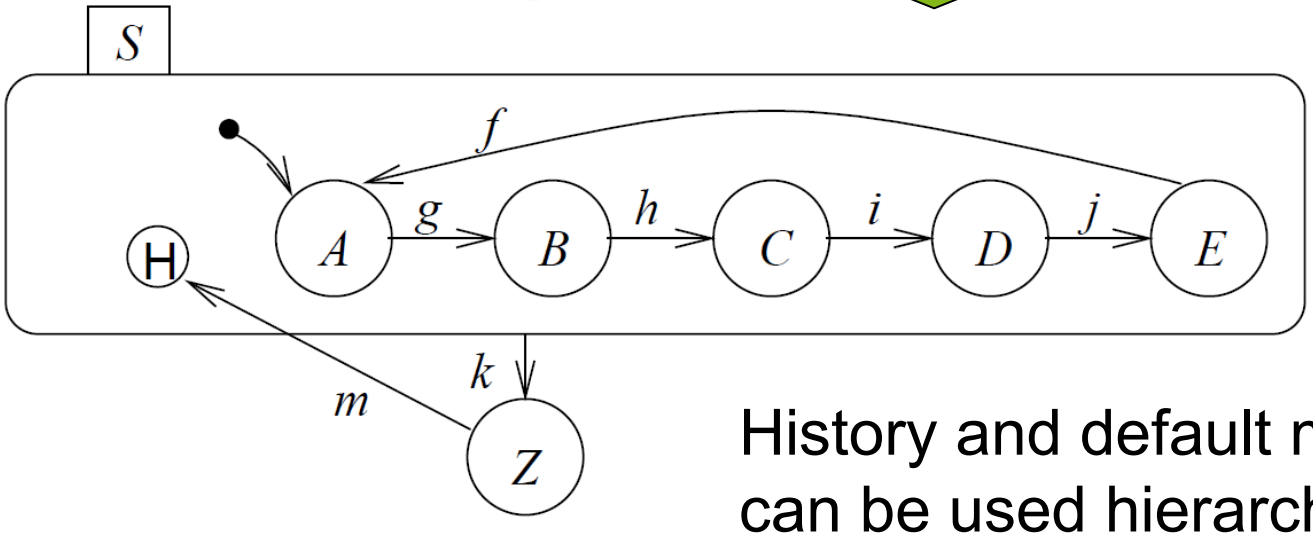
For input m, S enters the state it was in before S was left (can be A, B, C, D, or E).

If S is entered for the first time, the default mechanism applies.

Combining history and default state mechanism



same meaning

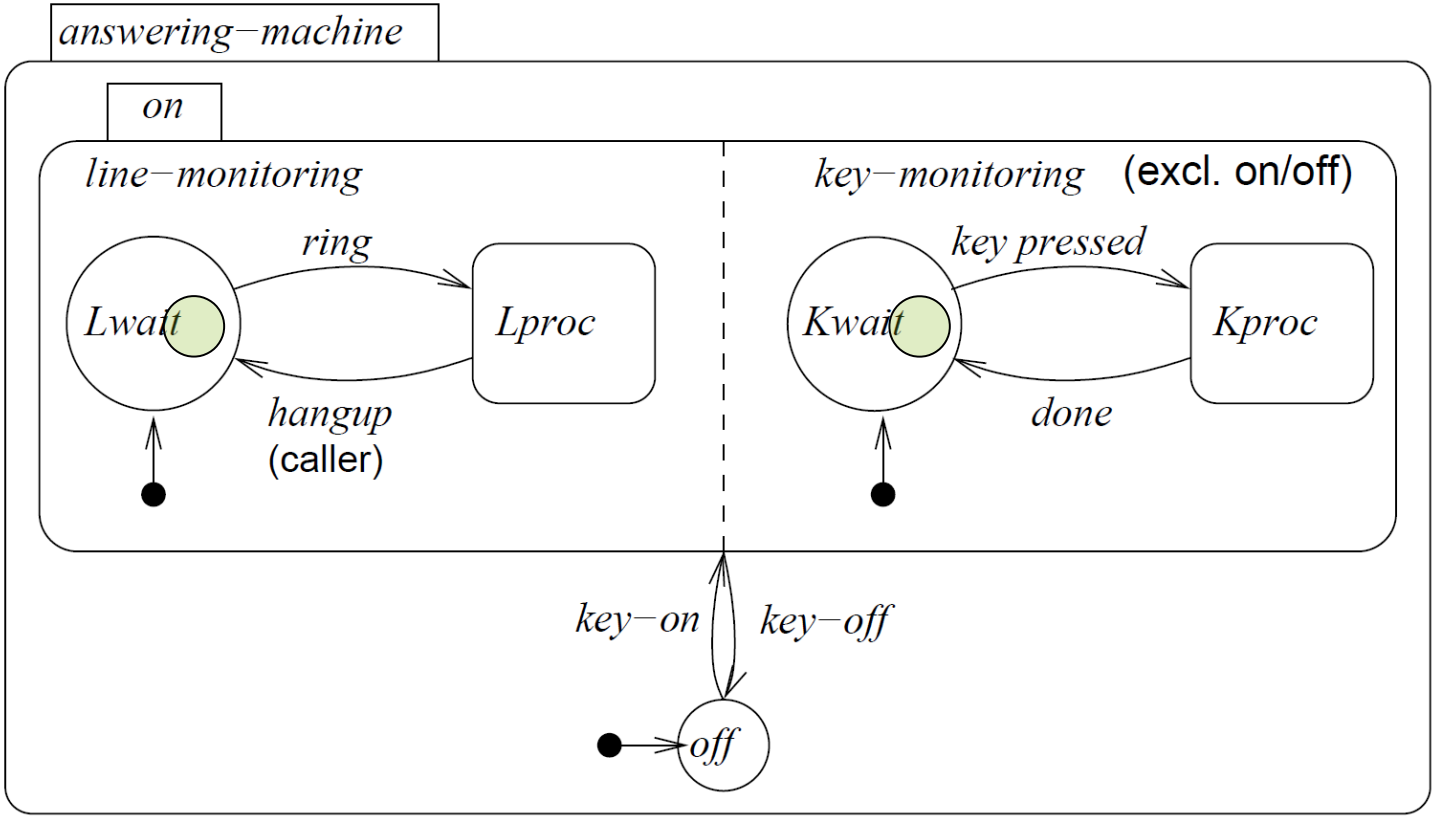


History and default mechanisms can be used hierarchically.

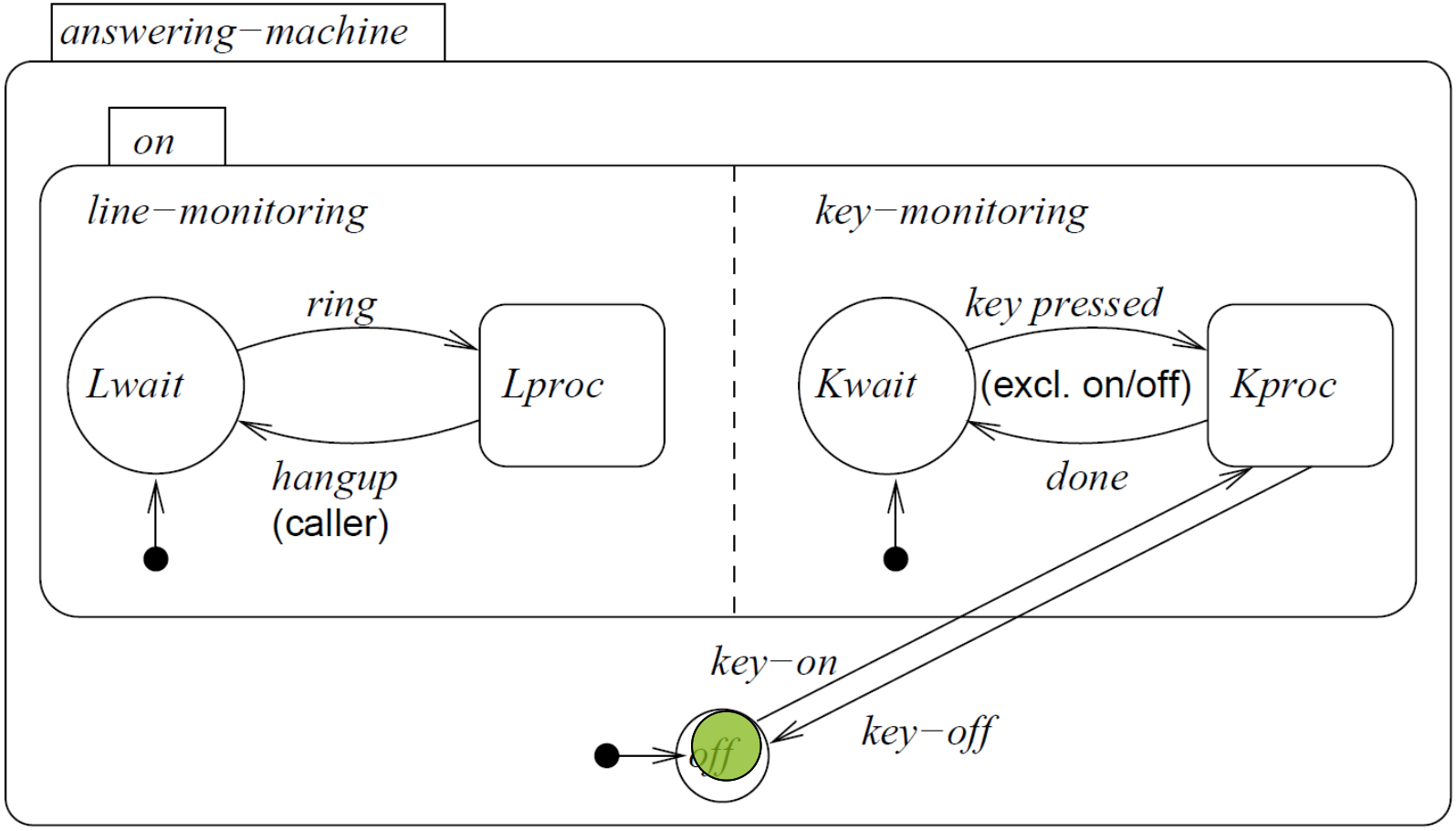
Concurrency

Convenient ways of describing concurrency req.
AND-super-states: FSM is in **all** (immediate) sub-states of a super-state

Example:



Entering and leaving AND-super-states



Line-monitoring and key-monitoring are entered and left, when service switch is operated.

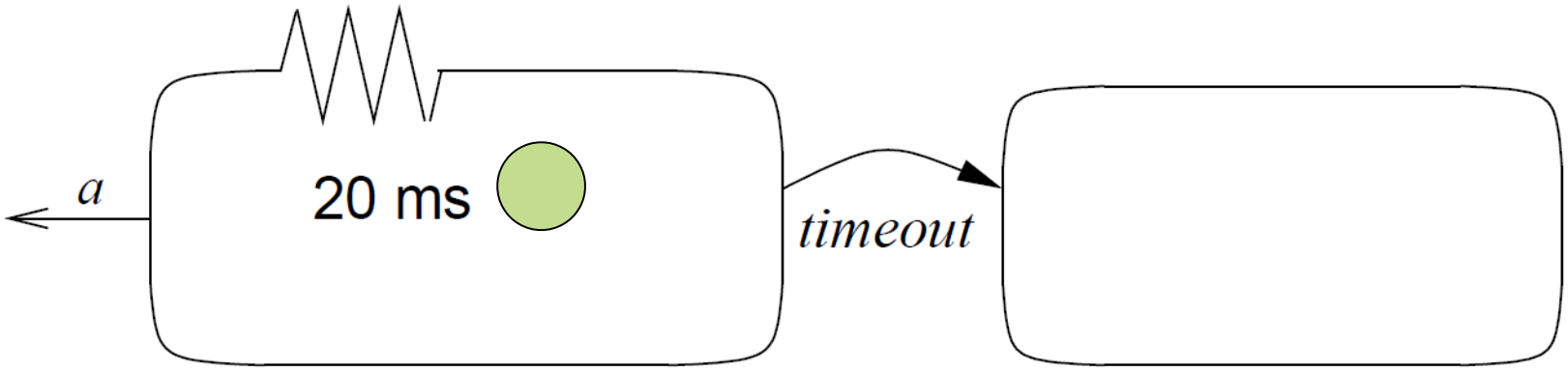
Types of states

In StateCharts, states are either

- **basic states**, or
- **AND-super-states**, or
- **OR-super-states**.

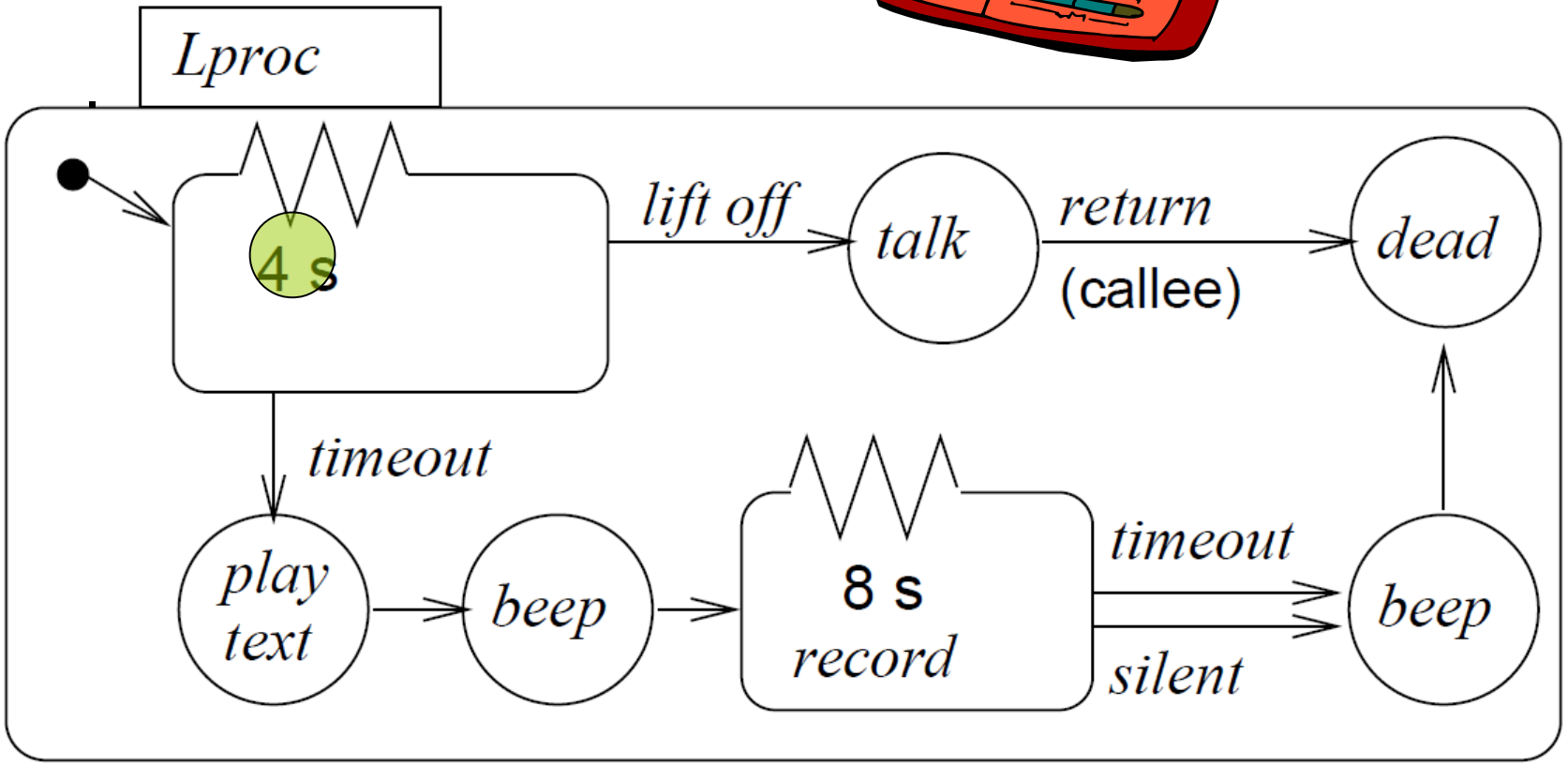
Timers

Since time needs to be modeled in embedded & cyber-physical systems, timers need to be modeled.
In StateCharts, special edges can be used for timeouts.

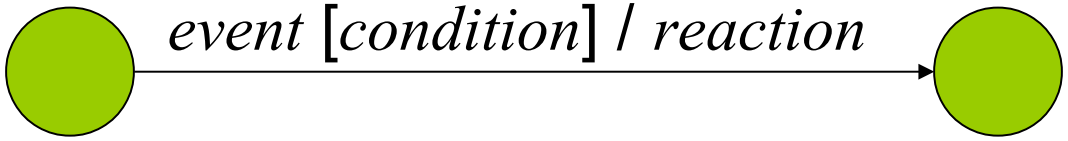


If event a does not happen while the system is in the left state for 20 ms, a timeout will take place.

Using timers in an answering machine



General form of edge labels



Events:

- Exist only until the next evaluation of the model
- Can be either internally or externally generated

Conditions:

- Refer to values of variables that keep their value until **they are reassigned**

Reactions:

- Can either be assignments for variables
- or creation of events

Example:

- *service-off* [not in *Lproc*] / *service*:=0

The StateCharts simulation phases (StateMate Semantics)

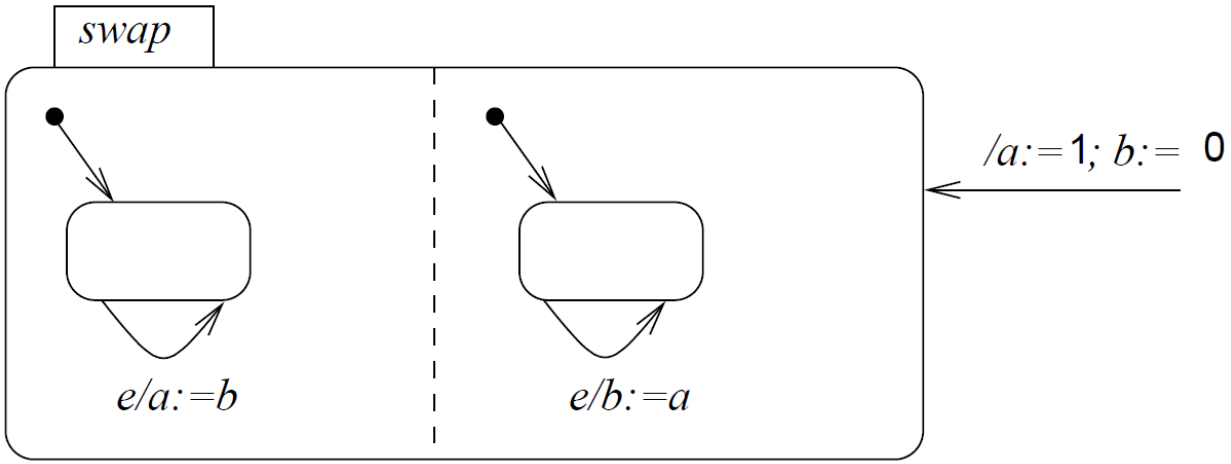
How are edge labels evaluated?

Three phases:

1. Effect of external changes on events and conditions is evaluated,
2. The set of transitions to be made in the current step and right hand sides of assignments are computed,
3. Transitions become effective, variables obtain new values.

Separation into phases 2 and 3 enables a resulting unique ("determinate") behavior.

Example (1)



In phase 2, variables a and b are assigned to temporary variables:

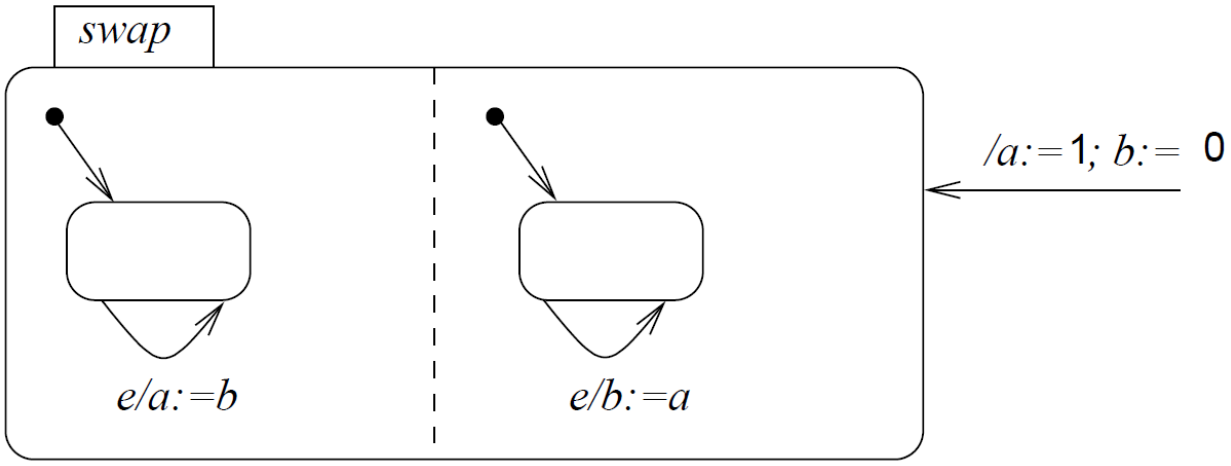
$$a' := b,\ b' := a;$$

In phase 3, these are assigned to a and b.

$$a := a',\ b := b';$$

As a result, variables a and b are swapped.

Example (2)



In a single phase environment, executing the left state first would assign the old value of b (=0) to a and b:

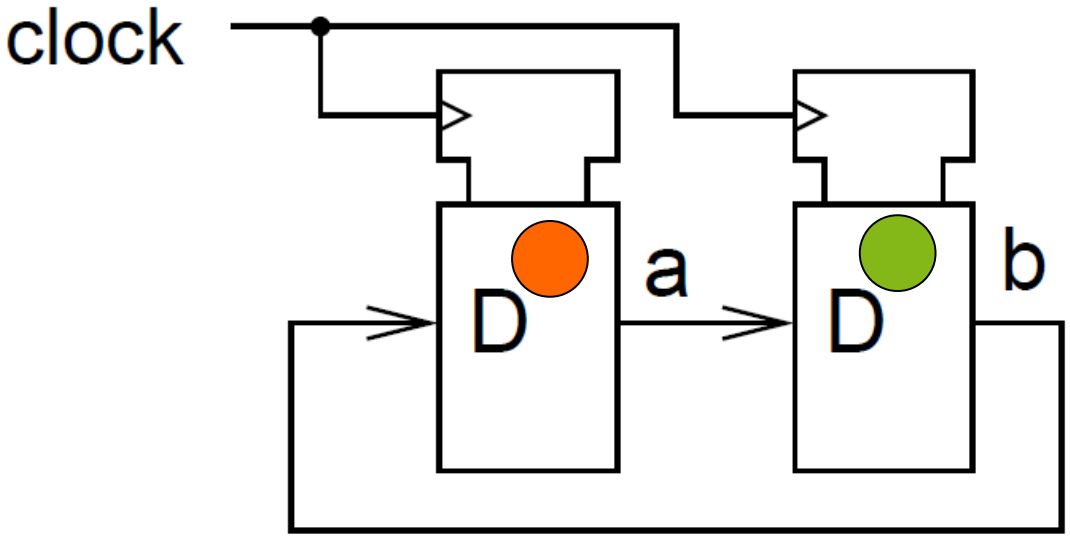
$a := 0, b := 0;$

Executing the right state first would assign the old value of a (=1) to a and b.

$b := 1, a := 1;$

The result would depend on the execution order.

Reflects model of clocked hardware

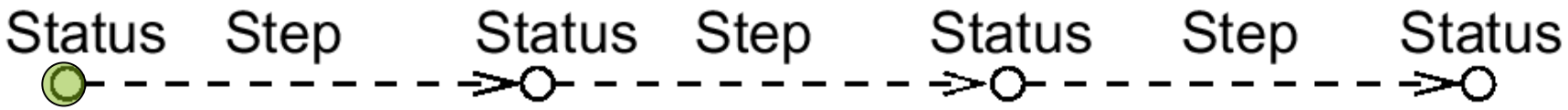


In an actual clocked (synchronous) hardware system, both registers would be swapped as well.

Same separation into phases found in other languages as well, especially those that are intended to model hardware.

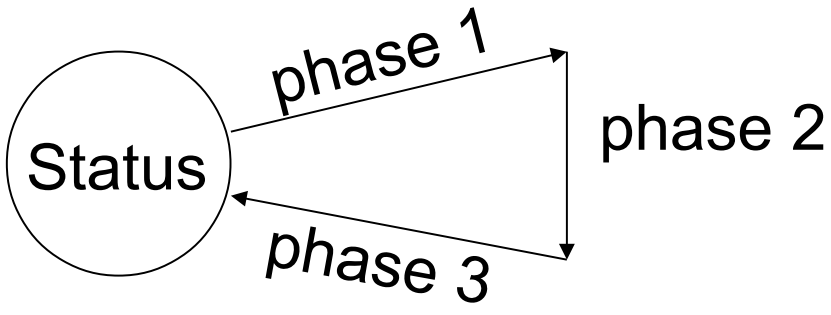
Steps

Execution of a StateMate model consists of a sequence of (status, step) pairs



Status = values of all variables + set of events + current time

Step = execution of the three phases (StateMate semantics)



Other implementations of StateCharts do not have these 3 phases (and hence could lead to different results)!

Other semantics

Several other specification languages for hierarchical state machines (UML, dave, …) do not include the three simulation phases.

These correspond more to a SW point of view with no synchronous clocks.

Some systems allow turning the multi-phased simulation on and off.

Broadcast mechanism

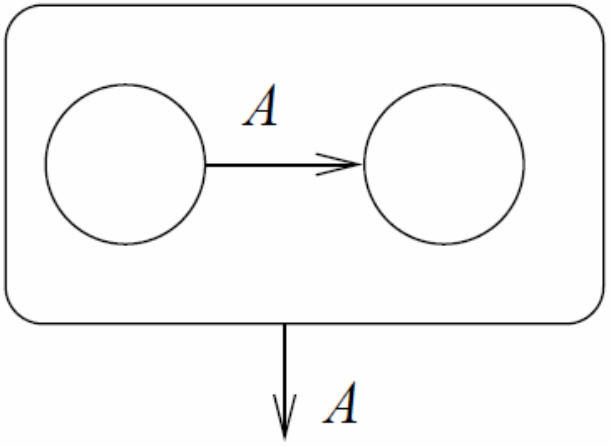
Values of variables are visible to all parts of the StateChart model
New values become effective in phase 3 of the current step and are obtained by all parts of the model in the following step. !

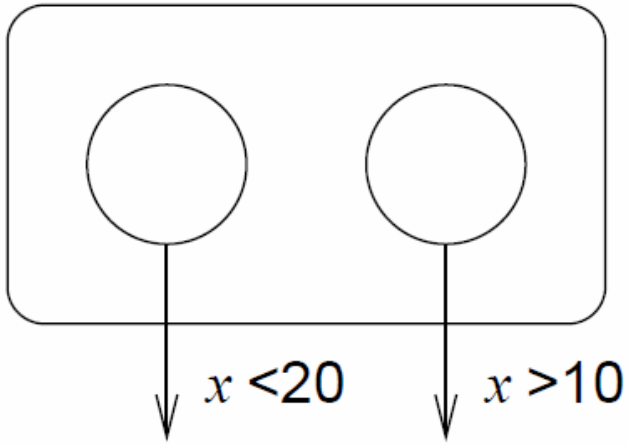
- ☞ StateCharts implicitly assumes a **broadcast** mechanism for variables
  (→ implicit ***shared memory communication***
  - other implementations would be very inefficient -).
- ☞ StateCharts is appropriate for local control systems (☺), but not for distributed applications for which updating variables might take some time (☹).

Lifetime of events

Events live until the step following the one in which they are generated (“one shot-events“).

Conflicts





Techniques for resolving these conflicts wanted

Determinate vs. deterministic

- Kahn (1974) calls a system **determinate** if we will always obtain the same result for a fixed set (and timing) of inputs
- Others call this property **deterministic**
  However, this term has several meanings:
  - Non-deterministic finite state machines
  - Non-deterministic operators
    (e.g. + with non-deterministic result in low order bits)
  - Behavior not known before run-time
    (unknown input results in non-determinism)
  - In the sense of determinate as used by Kahn

In order to avoid confusion, we use the term “determinate” in this course.

StateCharts determinate or not?

Must all simulators return the same result for a given input?

- Separation into 2 phases a required condition
- Semantics ≠ StateMate semantics may be non-determinate

Potential other sources of non-determinate behavior:

- Choice between conflicting transitions resolved arbitrarily: Tools typically issue a warning if such a situation could exist

→ **Determinate behavior for StateMate semantics if transition conflicts are resolved and no other sources of undefined behavior exist**

Evaluation of StateCharts (1)

PROs (👍):

- Hierarchy allows arbitrary nesting of AND- and OR-super states.
- (StateMate-) Semantics defined in a follow-up paper to original paper.
- Large number of commercial simulation tools available (StateMate, StateFlow, BetterState, ...)
- Available “back-ends” translate StateCharts into SW or HW languages, thus enabling software or hardware implementations.

Evaluation of StateCharts (2)

CONs (👎):

- Not useful for distributed applications,
- no program constructs,
- no description of non-functional behavior,
- no object-orientation,
- no description of structural hierarchy,
- generated programs may be inefficient.

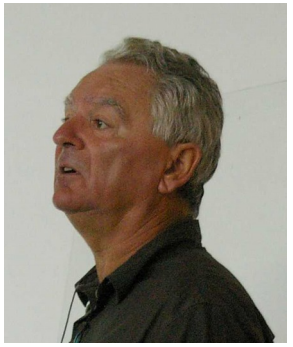
Extensions:

- Module charts for description of structural hierarchy.

Synchronous vs. asynchronous languages (1)

Description of several processes in many languages non-determinate: The order in which executable threads are executed is not specified (may affect result).

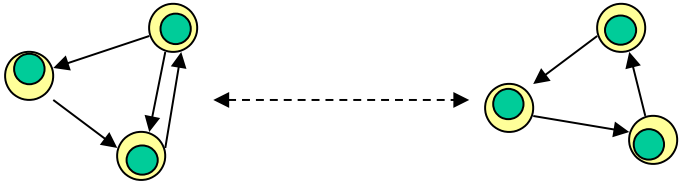
Synchronous languages: based on automata models. "*Synchronous languages aim at providing high level, modular constructs, to make the design of such an automaton easier*" [Nicolas Halbwachs].

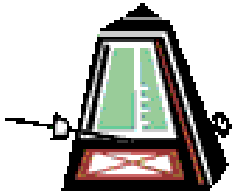


© P. Marwedel, 2008

Synchronous languages describe concurrently operating automata. "... *when automata are composed in parallel, a transition of the product is made of the "simultaneous" transitions of all of them".*

Synchronous vs. asynchronous languages (2)





Synchronous languages implicitly assume the presence of a (global) clock.

Each clock tick, all inputs are considered, new outputs and states are calculated and then the transitions are made.

Abstraction of delays

Let

- $f(x)$: some function computed from input x,
- $\Delta(f(x))$: the delay for this computation
- δ: some abstraction of the real delay

Consider compositionality: $f(x)=g(h(x))$

Then, the sum of the delays of g and h would be a safe upper bound on the delay of f.

Two solutions: 1. δ =0, always ☞ synchrony
2. δ =? (hopefully bounded) ☞ asynchrony

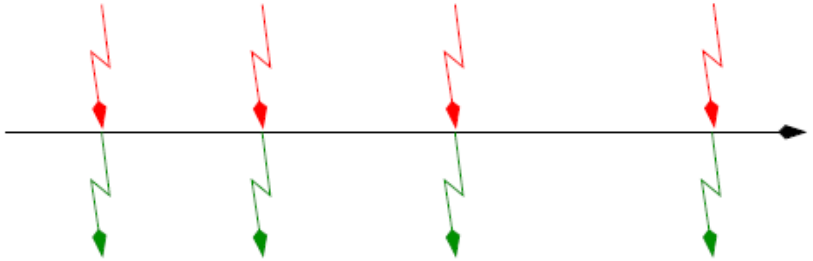
"Asynchronous languages don't work" [Halbwachs]
(Examples based on missing link to real time, e.g. what exactly does a **wait**(10 ns) in a programming language do?)

Based on slide 15 of N. Halbwachs: Synchronous Programming of Reactive Systems, *ARTIST2 Summer School on Embedded Systems*, Florianopolis, 2008

Compositionality

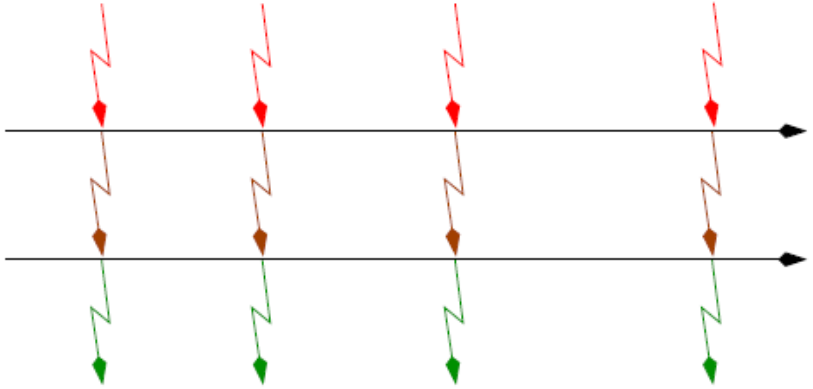
Abstract synchronous behavior

sequence of reactions to input events, to which all processes take part:



At the abstract level, a single FSM reacts ***immediately***

Composition of behaviors:



At the abstract level, reaction of connected other automata is ***immediate***

Based on slide 16 of N. Halbwachs: Synchronous Programming of Reactive Systems, *ARTIST2 Summer School on Embedded Systems*, Florianopolis, 2008

Concrete Behavior

The abstraction of synchronous languages is valid, as long as real delays are always shorter than the clock period.

Based on slide 17 of N. Halbwachs: Synchronous Programming of Reactive Systems, *ARTIST2 Summer School on Embedded Systems*, Florianopolis, 2008

Synchronous languages

- Require a broadcast mechanism for all parts of the model.
- Idealistic view of concurrency.
- Have the advantage of guaranteeing determinate behavior.
- *“StateCharts* (using StateMate semantics) *is an "almost" synchronous language”* [Halbwachs]. Immediate communication is the lacking feature which would make StateCharts a fully synchronous language.

Implementation and specification model

For synchronous languages, the **implementation** model is that of finite state machines (FSMs).

The specification may use different notational styles

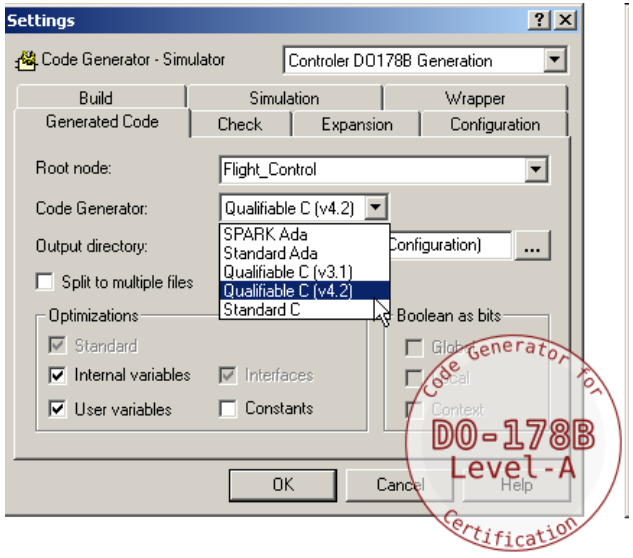
- “Imperative”: Esterel (textual)
- SyncCharts: graphical version of Esterel
- “Data-flow”: Lustre (textual)
- SCADE (graphical) is a mix containing elements from multiple styles

Nevertheless, specifications always include a close link to the generated FSMs (i.e., “imperative” does not have semantics close to von-Neumann languages)

Applications

SCADE Suite, including the SCADE KCG Qualified Code Generator, is used by AIRBUS and many of its main suppliers for the development of most of the A380 and A400M critical on board software, and for the A340-500/600 Secondary Flying Command System, aircraft in operational use since August 2002.

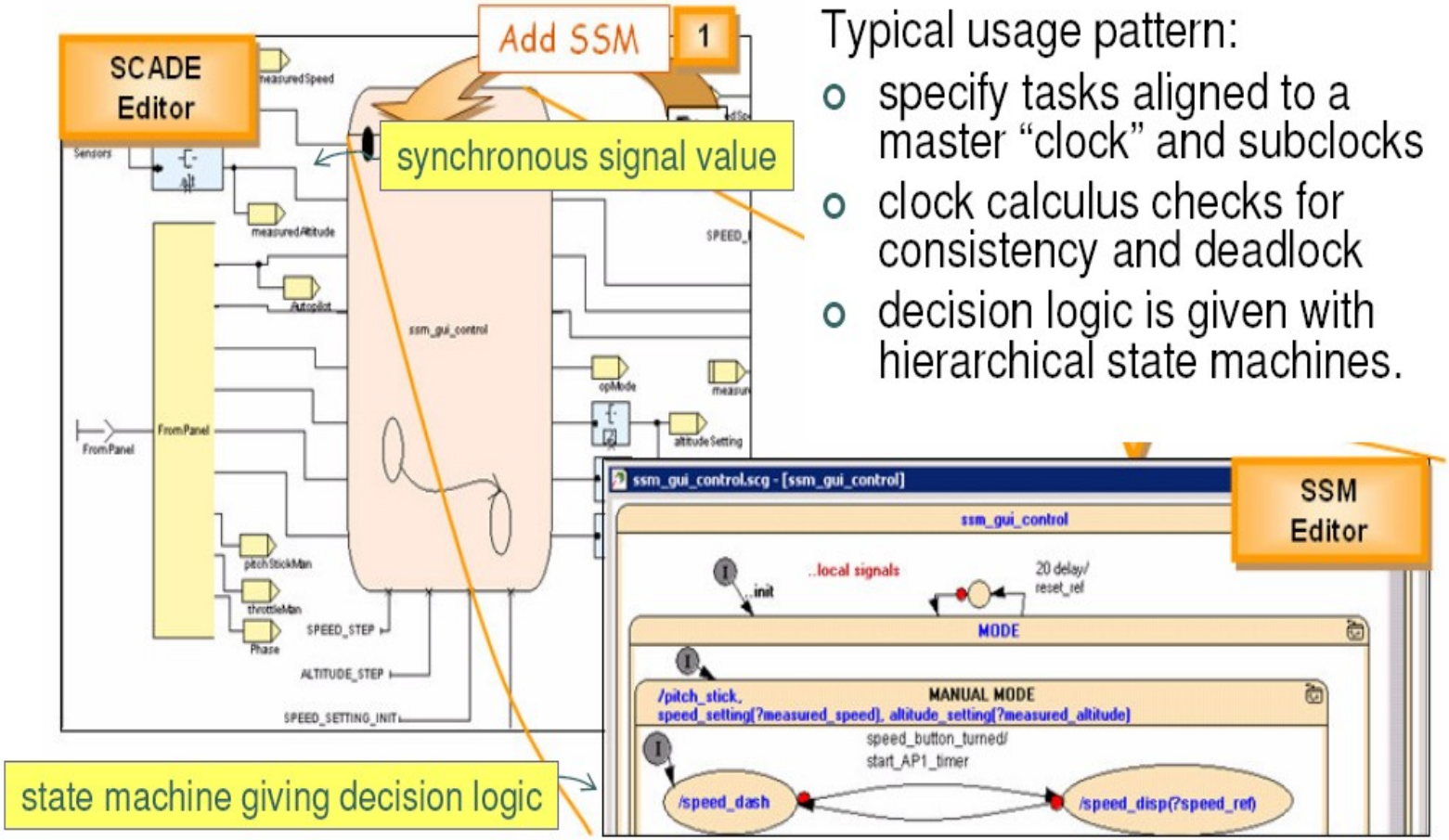
François Pilarski, Systems Engineering Framework - Senior Manager Engineering,Systems & Integration Tests; Airbus France.



Instance of "model-based design"

Source:
http://www.esterel-technologies.com/products/scade-suite/

Threads are Not the Only Possibility: 4th example: Synchronous Languages



Typical usage pattern:

- specify tasks aligned to a master "clock" and subclocks
- clock calculus checks for consistency and deadlock
- decision logic is given with hierarchical state machines.

Lustre/SCADE, from http://www.esterel-technologies.com/

Summary

Communicating finite state machines

- StateCharts
  - Hierarchical states
    - OR-States
    - AND-States
  - Timers
  - Broadcasting of updates of variables ☞ shared memory
  - Determinate vs. deterministic
- Synchronous languages
  - Based on clocked finite state machine view
  - Based on 0-delay (real delays must be small)

technische universität
dortmund

fakultät für informatik
informatik 12

FSMs & message passing: SDL

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 10 月 30 日

Models of computation considered in this course

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorith. ☞ Computer arch.		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC* …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

SDL (Specification and Description Language)

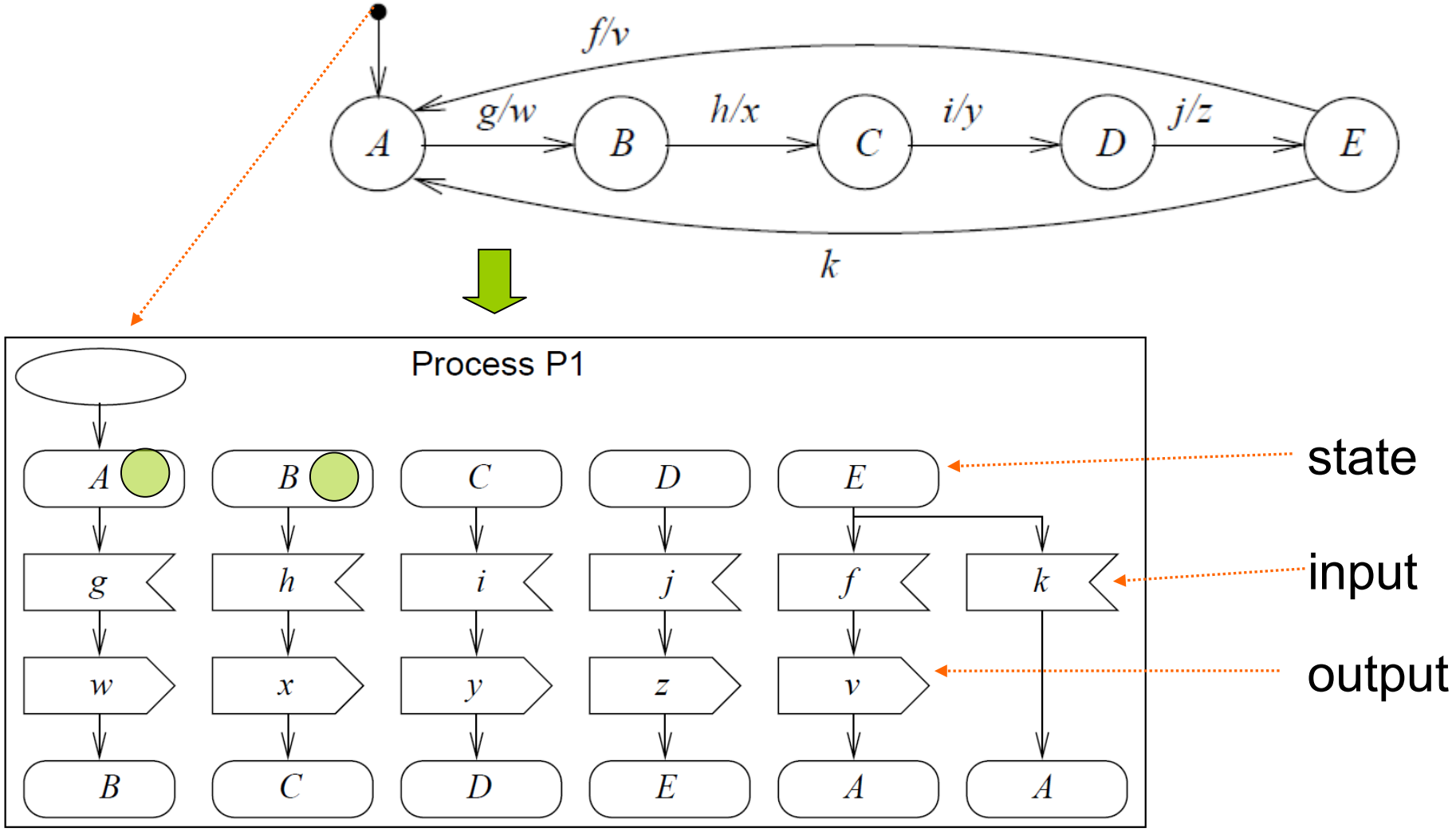
- SDL used here as a (prominent) example of a model of computation based on **asynchronous message passing communication**.

☞ SDL is appropriate also for distributed systems.

- Just like StateCharts, it is based on the CFSM model of computation; each FSM is called a **process**.

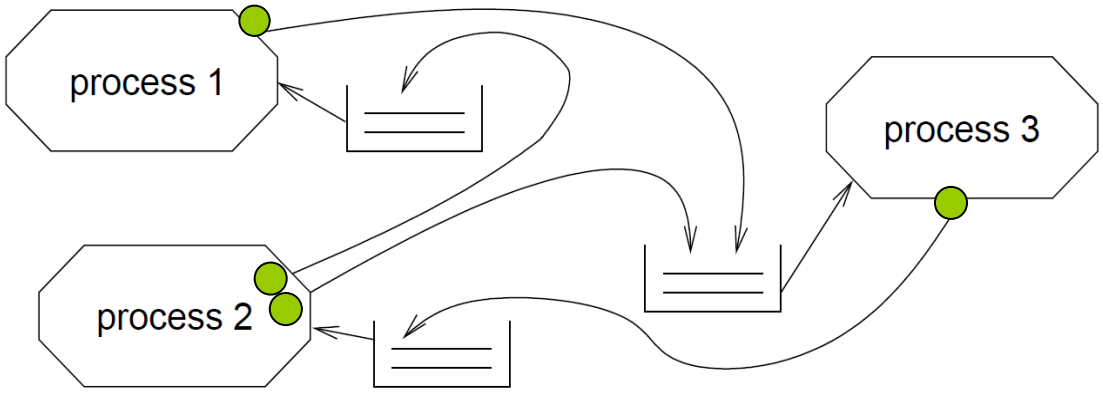
- Provides textual and graphical formats to please all users.

SDL-representation of FSMs/processes



Communication among SDL-FSMs

Communication is based on message-passing of *signals* (=inputs+outputs), assuming a **potentially indefinitely large FIFO-queue**.

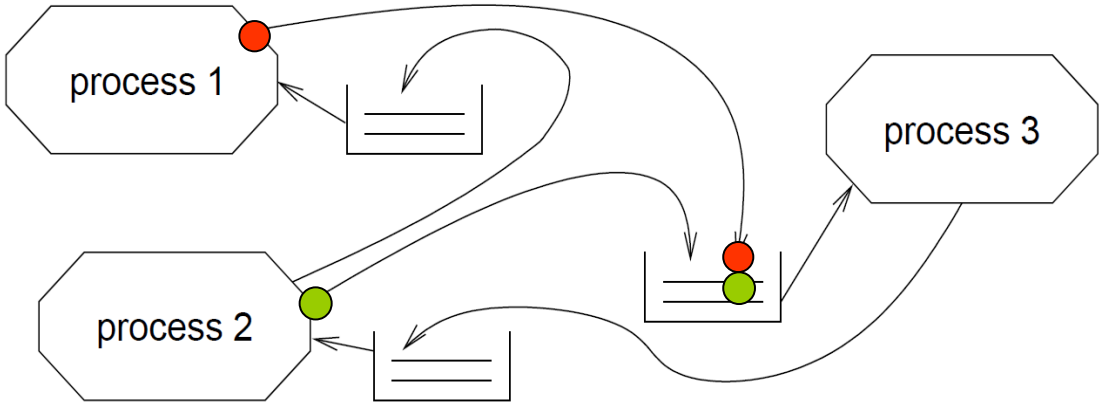


- Each process fetches next signal from FIFO,
- checks if signal enables transition,
- if yes: transition takes place,
- if no: signal is ignored (exception: SAVE-mechanism).
- Implementation requires bound for the maximum length of FIFOs

Determinate?

Let signals be arriving at FIFO at the same time:

- ☞ Order in which they are stored, is unknown

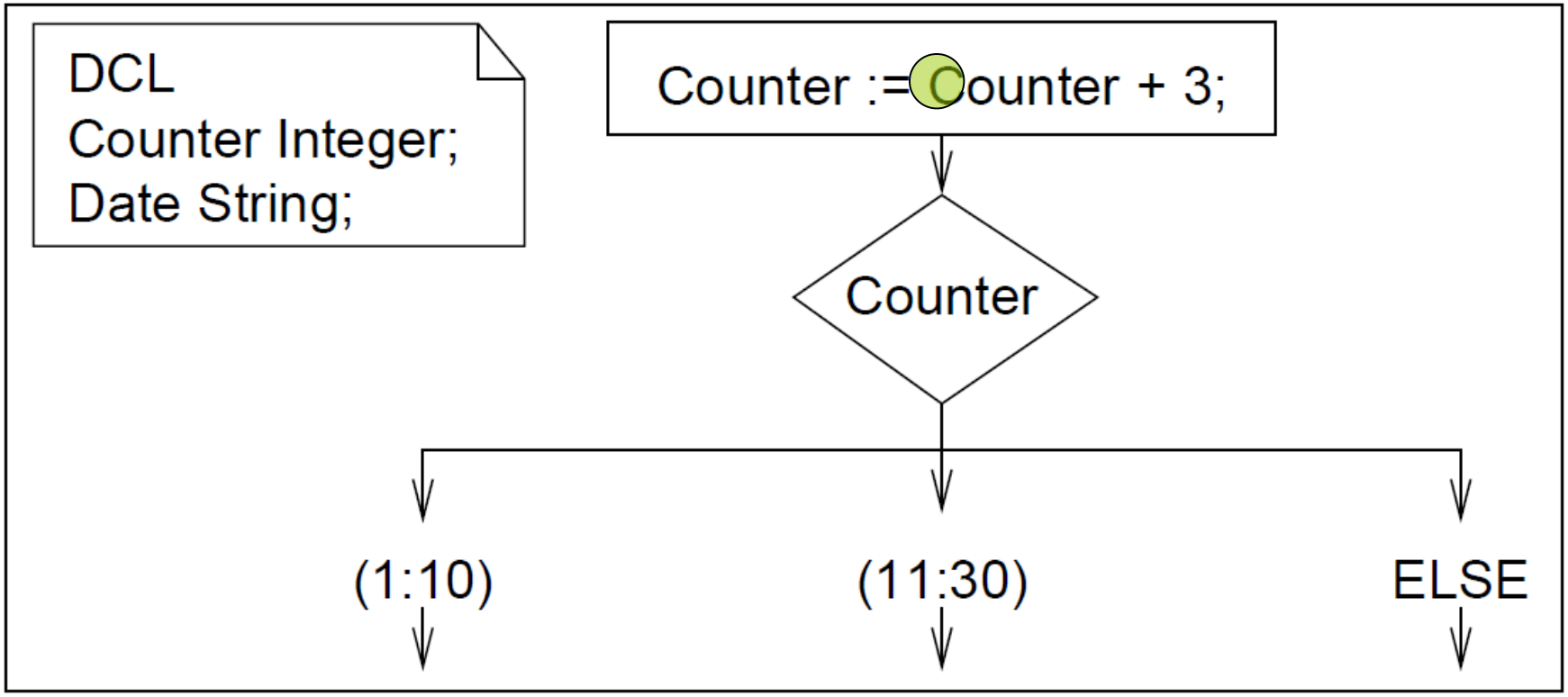


All orders are legal:

- ☞ simulators can show different behaviors for the same input, all of which are correct.

Operations on data

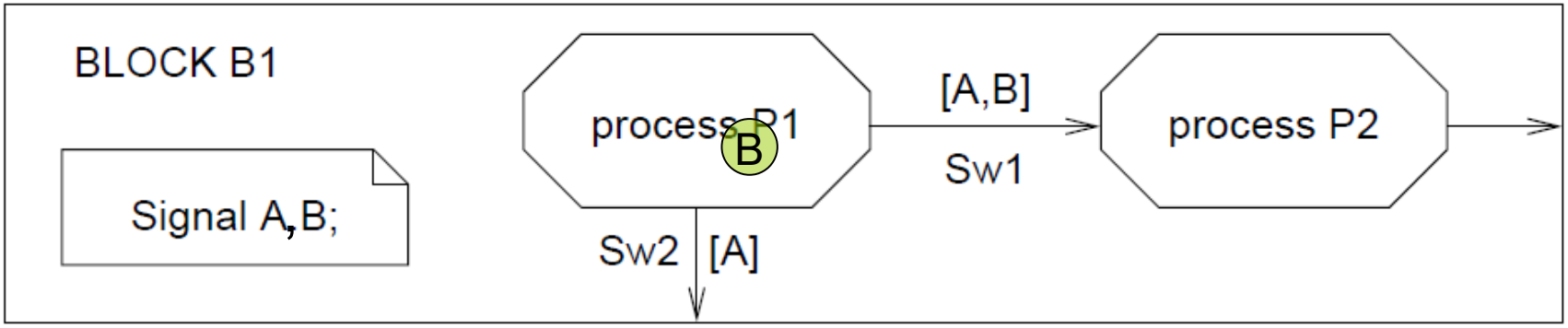
Variables can be declared locally for processes.
Their type can be predefined or defined in SDL itself.
SDL supports abstract data types (ADTs). Examples:



Process interaction diagrams

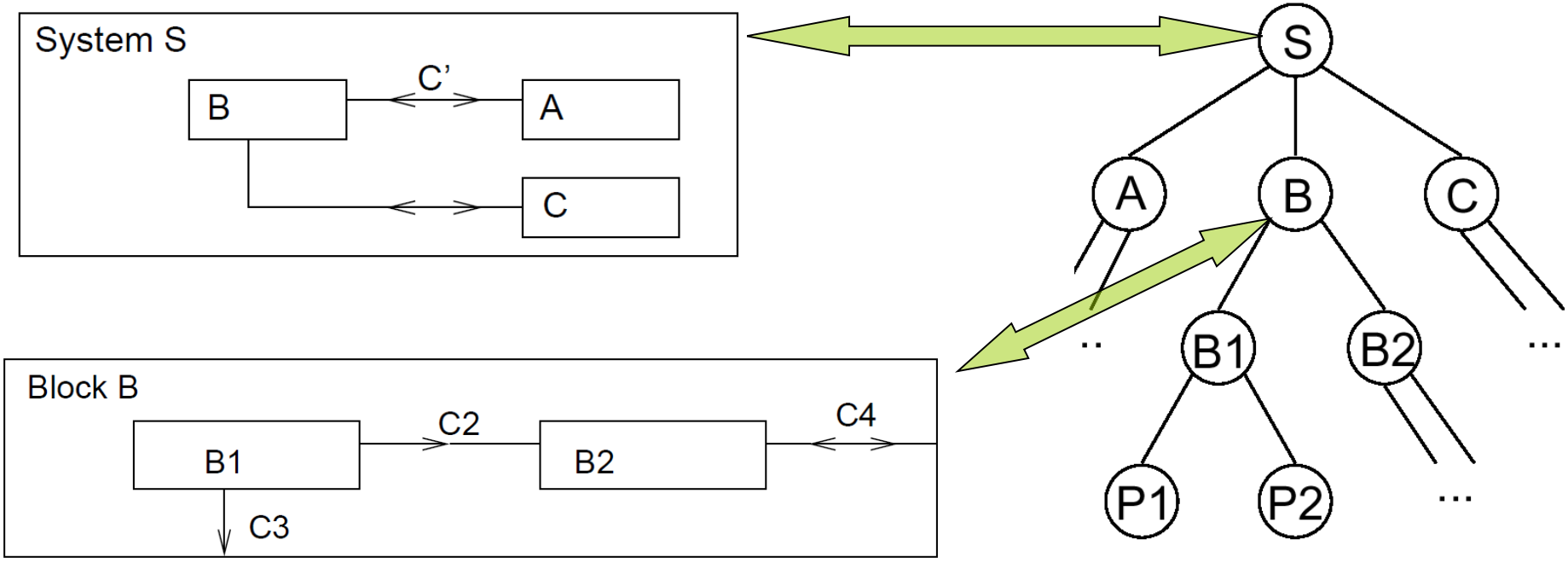
Interaction between processes can be described in process interaction diagrams (special case of block diagrams).
In addition to processes, these diagrams contain channels and declarations of local signals.

Example:



Hierarchy in SDL

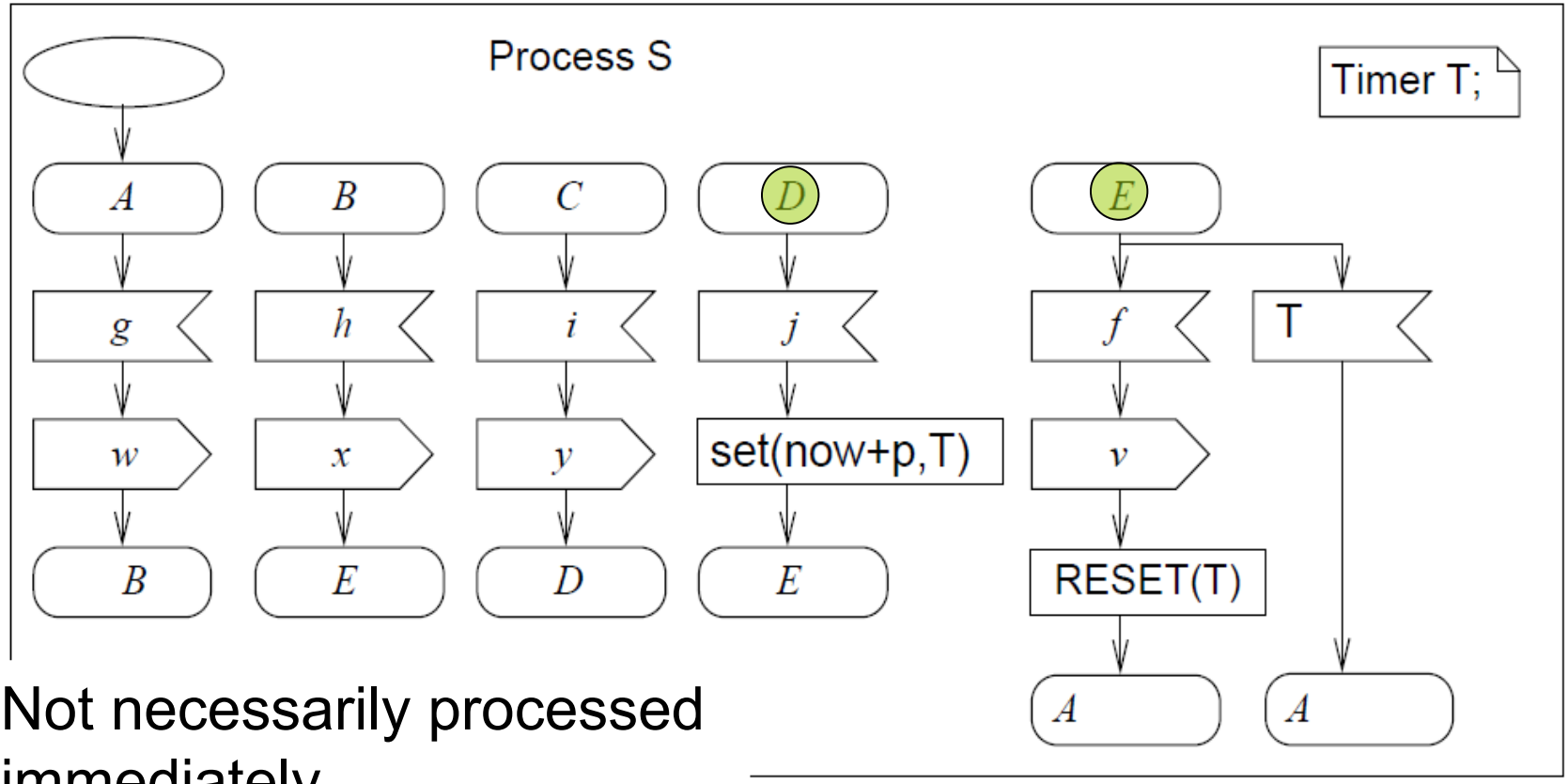
Process interaction diagrams can be included in **blocks.**
The root block is called **system.**



Processes cannot contain other processes, unlike in StateCharts.

Timers

Timers can be declared locally. Elapsed timers put signal into queue. RESET removes timer (also from FIFO-queue).



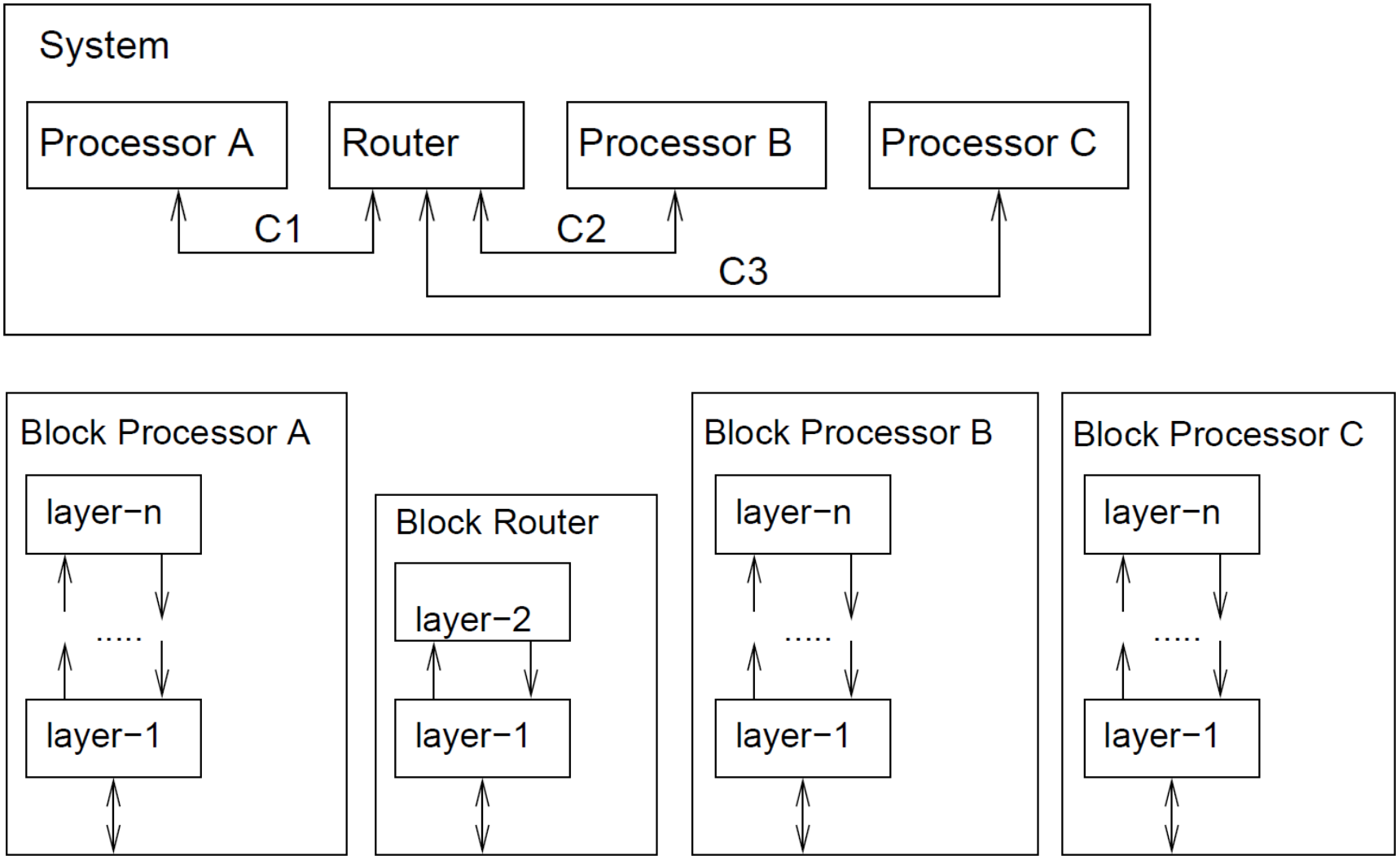
Not necessarily processed immediately.

Additional language elements

SDL includes a number of additional language elements, like

- procedures
- creation and termination of processes
- advanced description of data
- more features added for SDL-2000

Application: description of network protocols

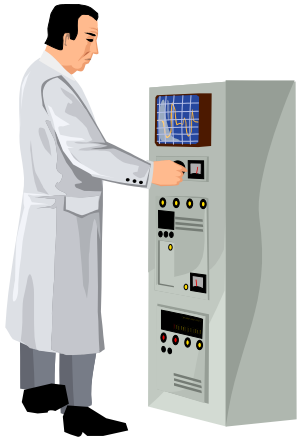


Larger example: vending machine

Machine° selling pretzels, (potato) chips, cookies, and doughnuts:

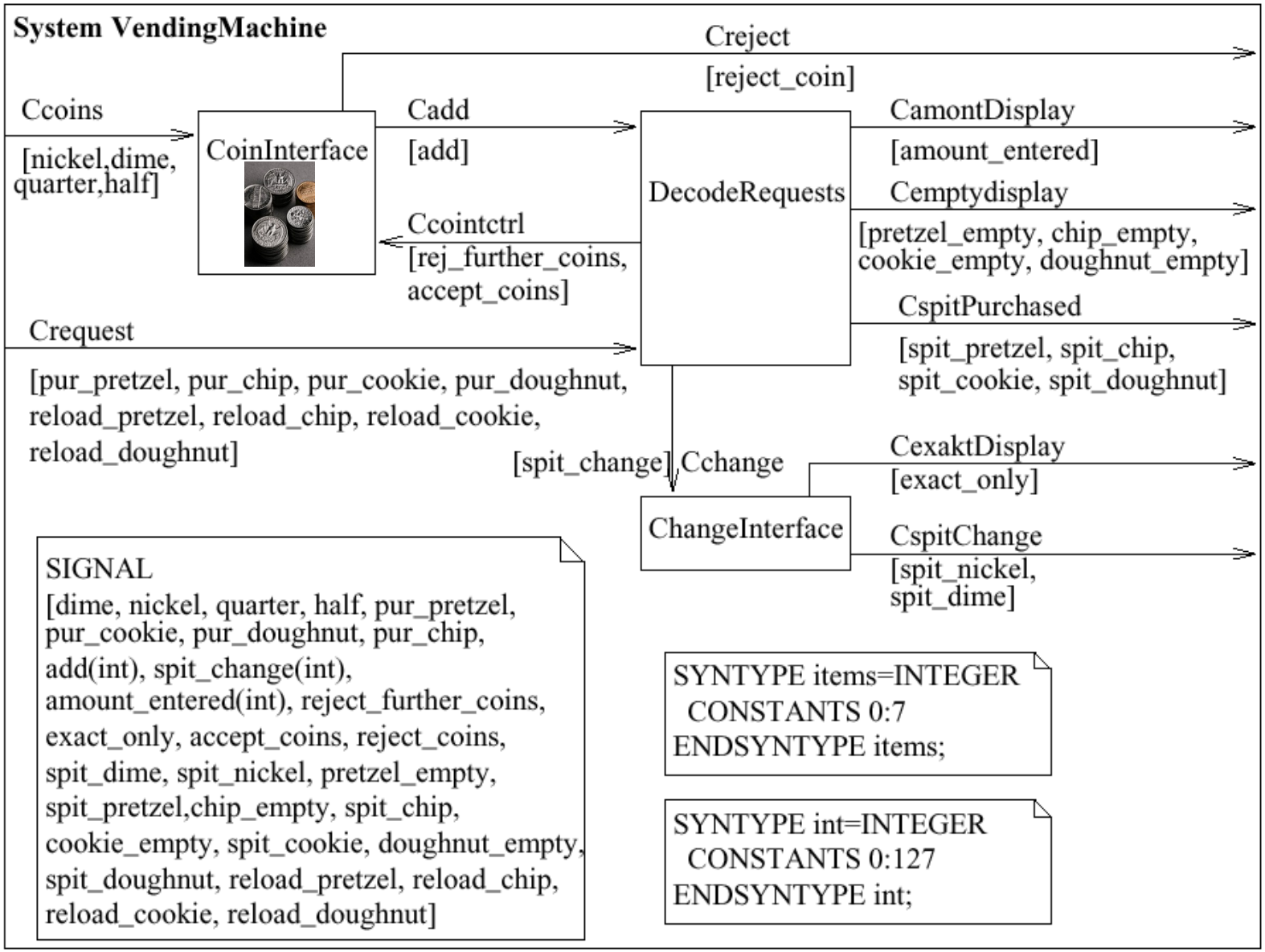
accepts nickels, dime, quarters, and half-dollar coins.

Not a distributed application.

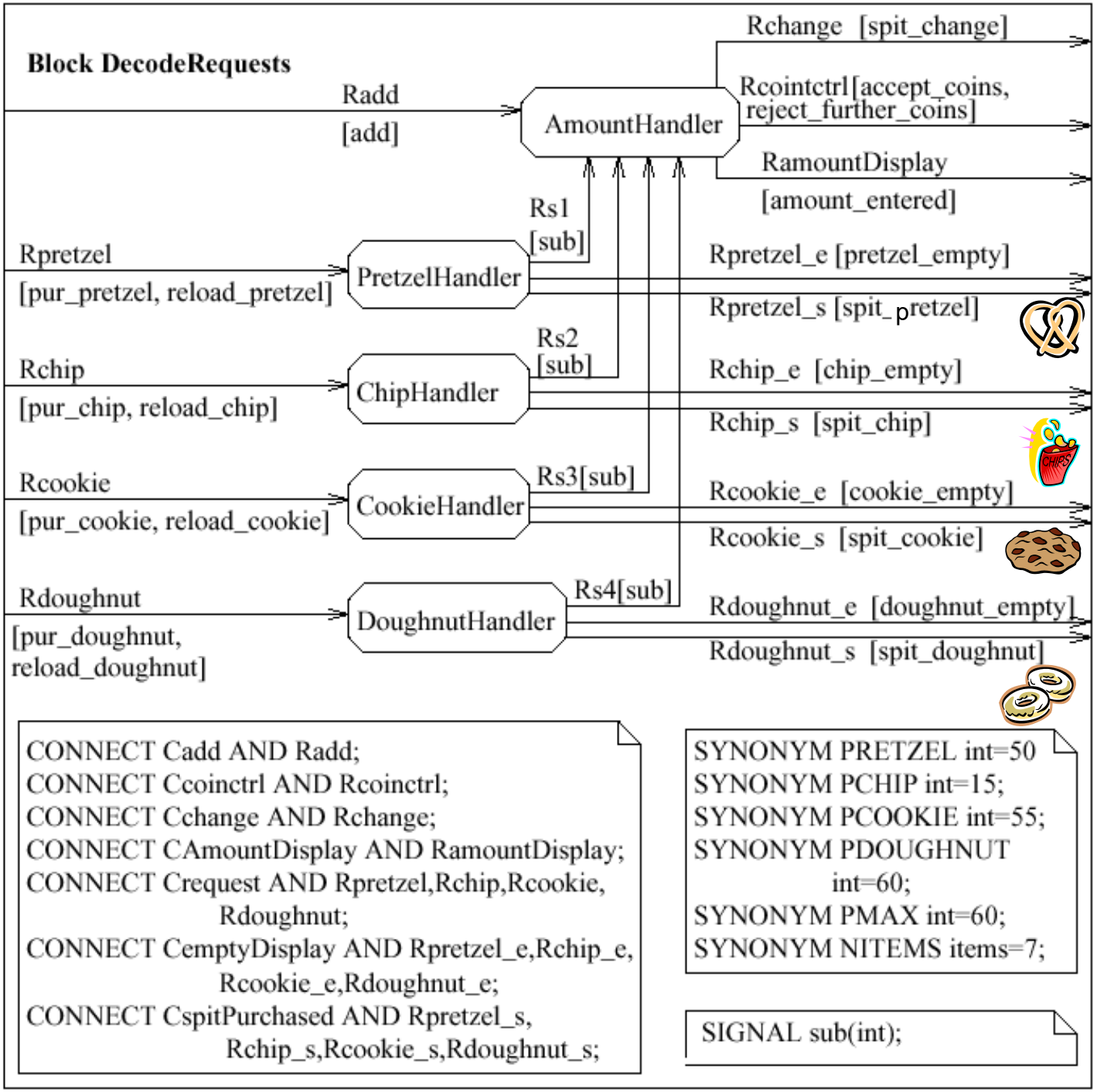


° J.M. Bergé, O. Levia, J. Roullard: High-Level System Modeling, Kluwer Academic Publishers, 1995

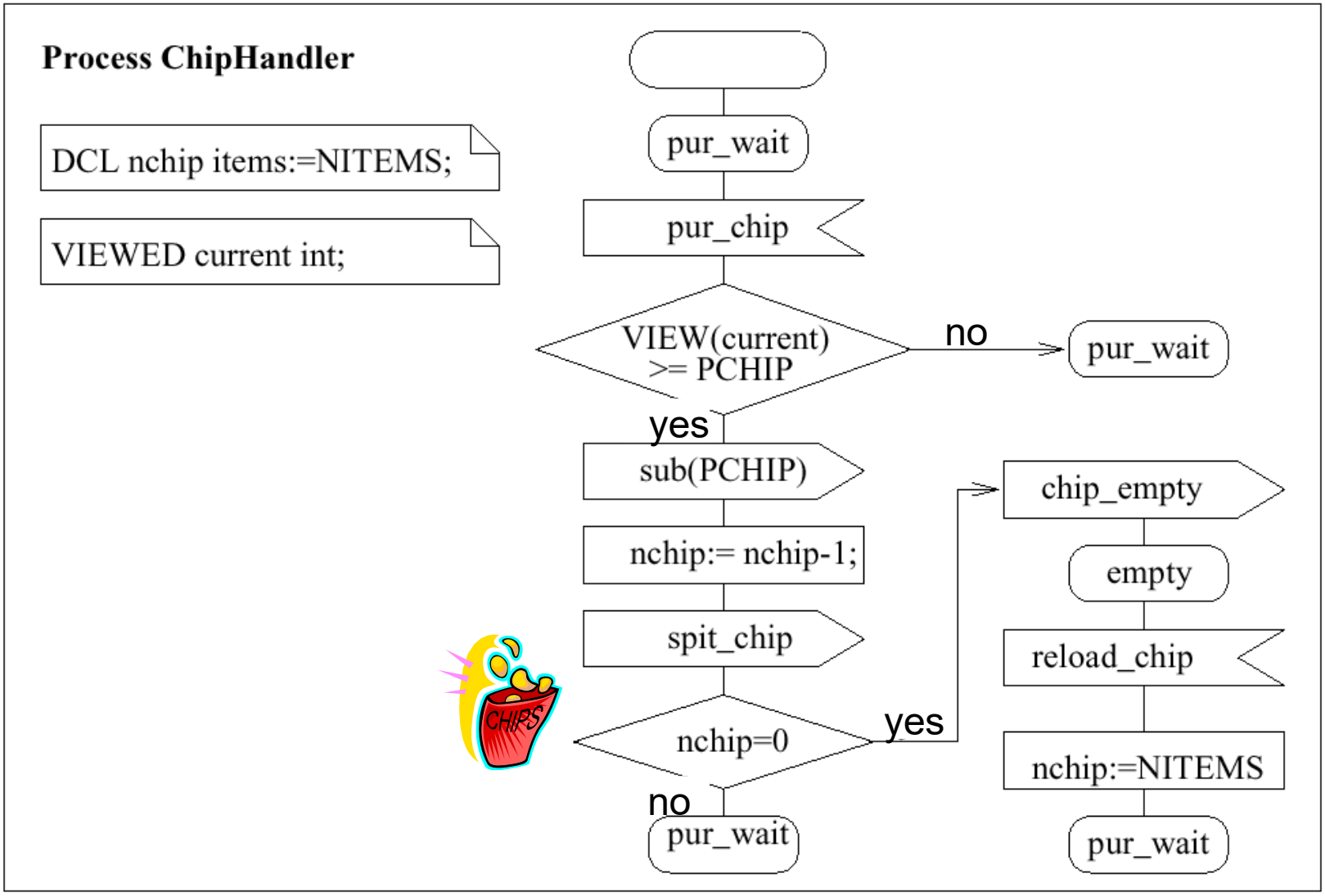
Overall view of vending machine



Decode Requests



ChipHandler



History

- Dates back to early 1970s,
- Formal semantics defined in the late 1980s,
- Defined by ITU (International Telecommunication Union):
  Z.100 recommendation in 1980
  Updates in 1984, 1988, 1992, 1996 and 1999
- SDL-2000 a significant update (not well accepted)
- Becoming less popular

Evaluation & summary

- FSM model for the components,
- Non-blocking message passing for communication,
- Implementation requires bound for the maximum length of FIFOs; may be very difficult to compute,
- Excellent for distributed applications (used for ISDN),
- Commercial tools available (see http://sdl-forum.org)
- Not necessarily determinate
- Timer concept adequate just for soft deadlines,
- Limited way of using hierarchies,
- Limited programming language support,
- No description of non-functional properties,
- Examples: small network + vending machine

Data flow models

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 10 月 30 日

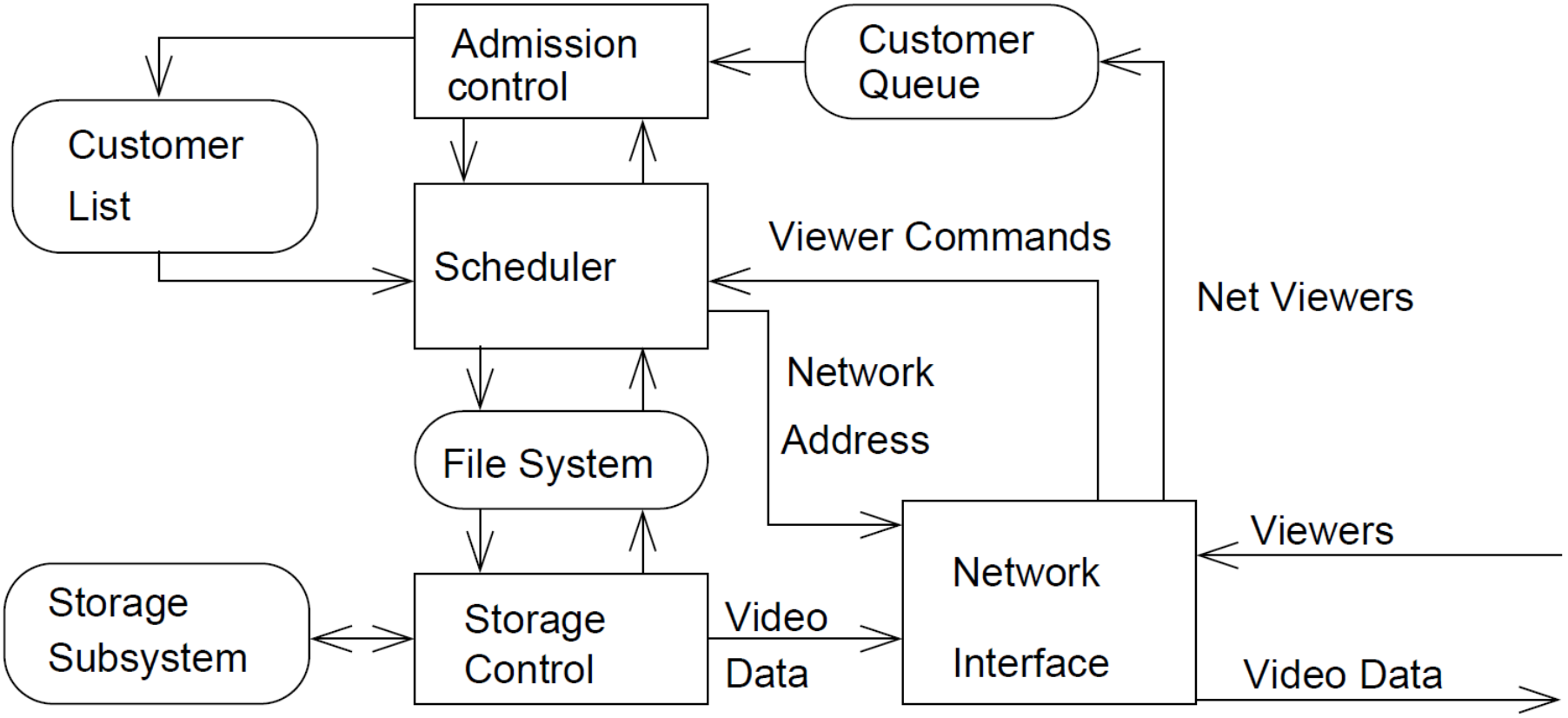
Models of computation considered in this course

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorith. ☞ Computer arch.		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC* …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

Data flow as a “natural” model of applications

Example: Video on demand system



www.ece.ubc.ca/~irenek/techpaps/vod/vod.html

Data flow modeling

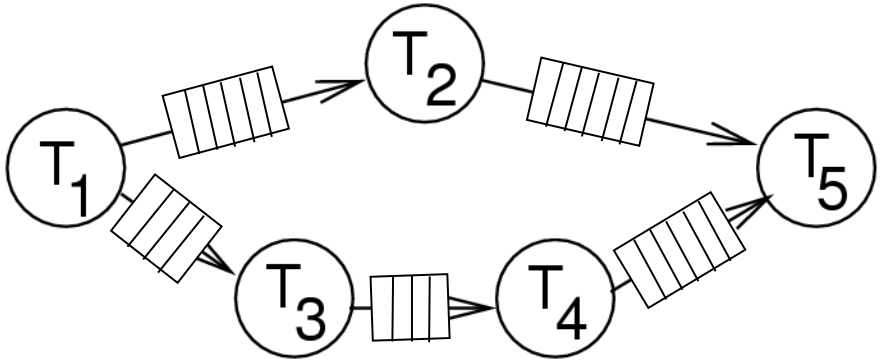
Definition: Data flow modeling is "...*the process of identifying, modeling and documenting how data moves around an information system.*
Data flow modeling examines

- *processes (activities that transform data from one form to another),*
- *data stores (the holding areas for data),*
- *external entities (what sends data into a system or receives data from a system, and*
- *data flows (routes by which data can flow)*".

Wikipedia: Structured systems analysis and design method.
https://en.wikipedia.org/wiki/Structured_systems_analysis_and_design_method

Kahn process networks (KPN)

- Each component is modeled as a program/task/process, (underlying FSM is inconvenient: possibly many states)
- Communication is by FIFOs; no overflow considered
  - ☞ writes never have to wait,
  - ☞ reads wait if FIFO is empty.



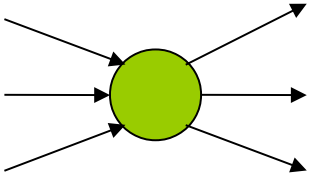
- Only one sender and one receiver per FIFO
  - ☞ no SDL-like conflicts at FIFOs

Example

```
process f(in int u, in int v, out int w){
   int i; bool b = true;
   for (;;) {
    i= b ? read(u) : read(v);
                    //read returns next token in FIFO, waits if empty
    send (i,w);  //writes a token into a FIFO w/o blocking
    b = !b;
    }
```

Properties of Kahn process networks

- Communication is only via channels (no shared variables)
- Mapping from ≥ 1 input channel to ≥ 1 output channel possible;

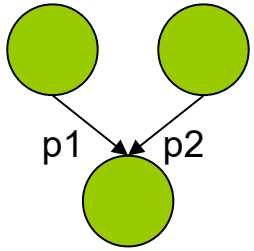


- Channels transmit information within an unpredictable but finite amount of time;
- In general, execution times are unknown.

Key beauty of KPNs (1)

- A process cannot check for available data before attempting a read (wait).

~~if nonempty(p1) then read(p1)~~
~~else if nonempty(p2) then read(p2);~~



- A process cannot wait for data for >1 port at a time.

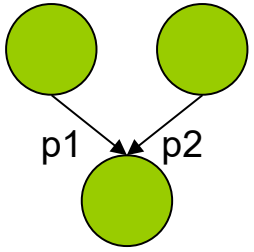
~~read(p1|p2);~~

☞ Processes have to commit to wait for data **from a particular port**;

Key beauty of KPNs (2)

- Therefore, the order of reads does not depend on the arrival time (but may depend on data).

- Therefore, Kahn process networks are determinate (!); for a given input, the result will always the same, regardless of the speed of the nodes.



- Many applications in embedded system design: Any combination of fast and slow simulation & hardware prototypes always gives the same result.

Computational power and analyzability

- It is a challenge to schedule KPNs without accumulating tokens
- KPNs are Turing-complete
  - We can implement a Turing machine as a KPN
  - However, the difficulty of Turing machines is also inherited
  - Analyzing the maximum FIFO size is “undecidable”
  - Analyzing whether it terminates is “undecidable”
  - Deadlock may be possible
- Timing is not specified
- Number of processes is static (cannot change)

More information about KPNs

- http://ls12-www.cs.tu-dortmund.de/teaching/download/levi/index.html: Animation
- https://en.wikipedia.org/wiki/Kahn_process_networks
- See also S. Edwards: http://www.cs.columbia.edu/~sedwards/classes/2001/w4995-02/presentations/dataflow.ppt

SDF

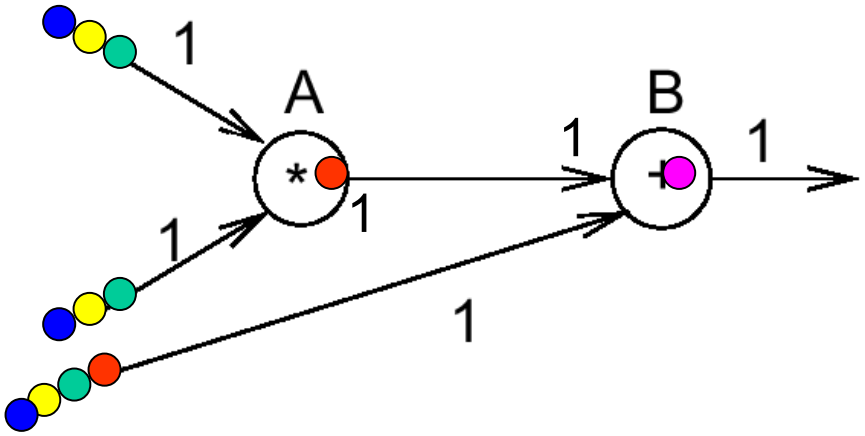
Less computationally powerful, but easier to analyze:

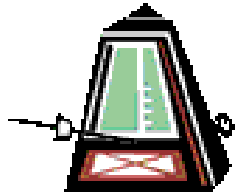
Synchronous data flow (SDF).

- **Synchronous**
  - all the tokens are consumed synchronously
  - one way to implement is to have a global clock controlling “firing” of nodes
- Again using **asynchronous** message passing
  = tasks do not have to wait until output is accepted.

(Homogeneous-) Synchronous data flow (SDF)

- Nodes are called ***actors.***
- Actors are ***ready***, if the necessary number of input tokens exists and if enough buffer space at the output exists
- Ready actors ***can fire*** (be executed).



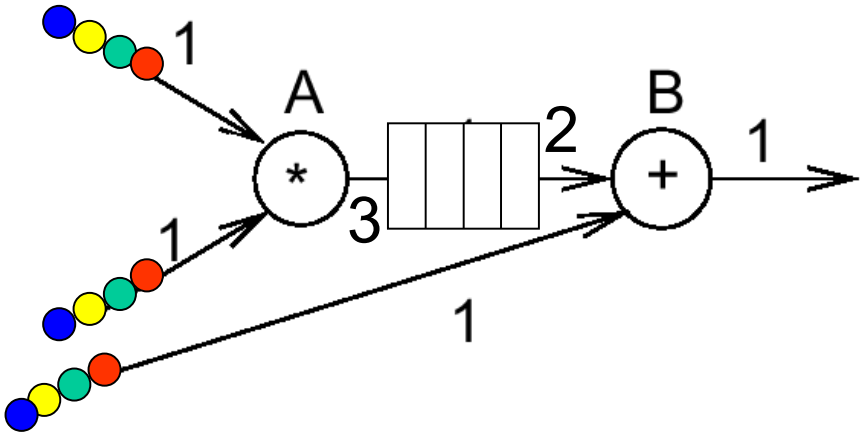


- Execution takes a ***fixed, known time***.

Actually, this is a case of ***homogeneous*** synchronous data flow models (HSDF): # of tokens per firing the same. ☞

(Non-homogeneous-) Synchronous data flow (SDF) (1)

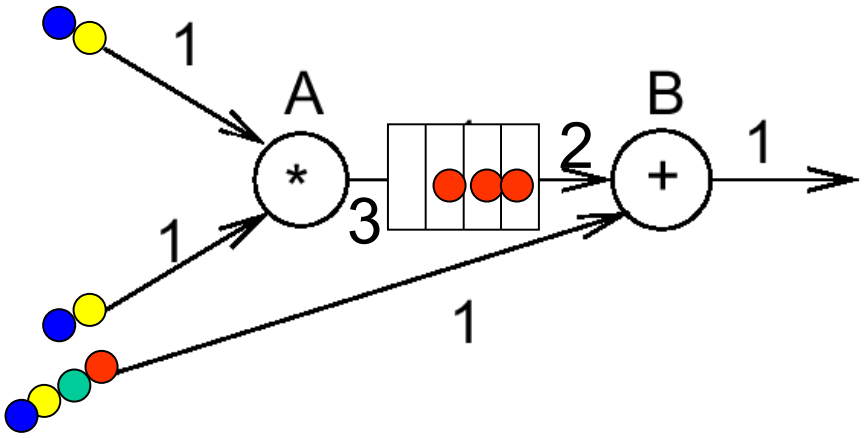
In the general case, a number of tokens can be produced / consumed per firing



A ready, can fire (does not have to)

(Non-homogeneous-) Synchronous data flow (SDF) (2)

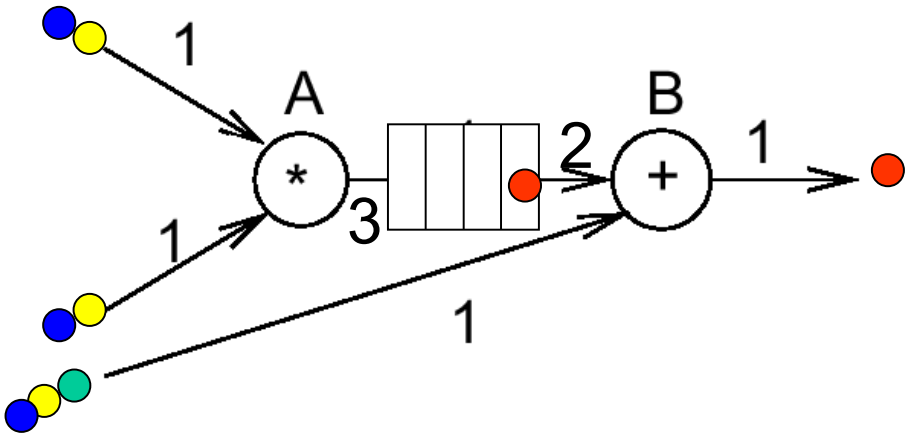
In the general case, a number of tokens can be produced / consumed per firing



B ready, can fire

(Non-homogeneous-) Synchronous data flow (SDF) (3)

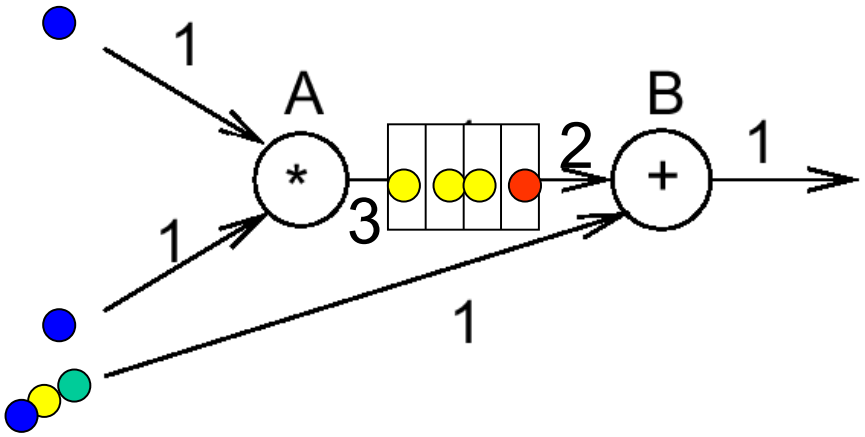
In the general case, a number of tokens can be produced / consumed per firing



A ready, can fire

(Non-homogeneous-) Synchronous data flow (SDF) (4)

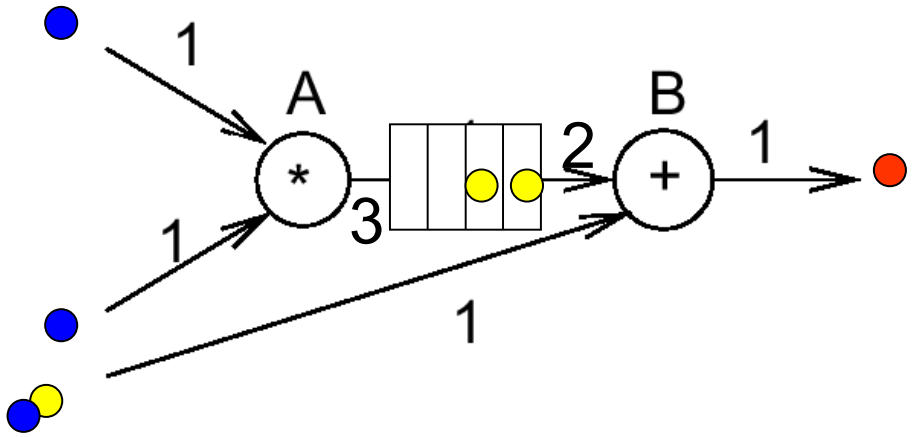
In the general case, a number of tokens can be produced / consumed per firing



B ready, can fire

(Non-homogeneous-) Synchronous data flow (SDF) (5)

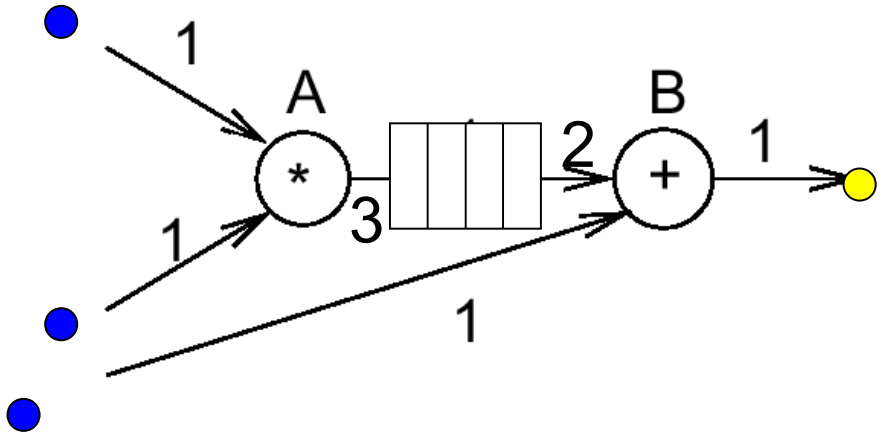
In the general case, a number of tokens can be produced / consumed per firing



B ready, can fire

(Non-homogeneous-) Synchronous data flow (SDF) (6)

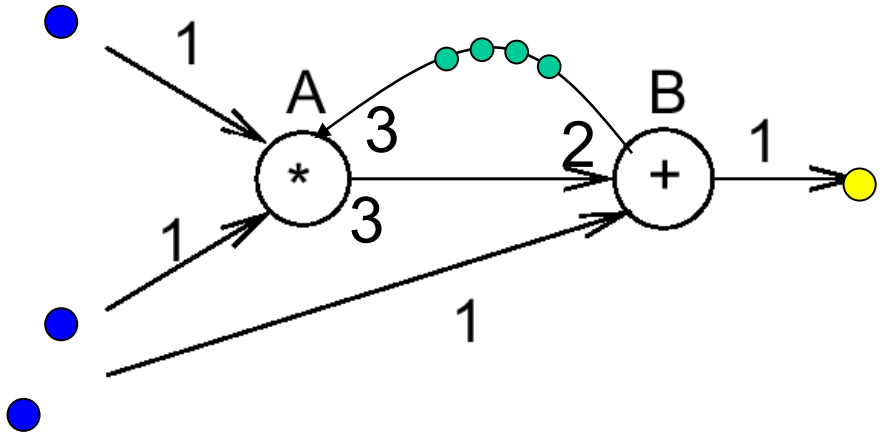
In the general case, a number of tokens can be produced / consumed per firing



1 period complete, A ready, can fire

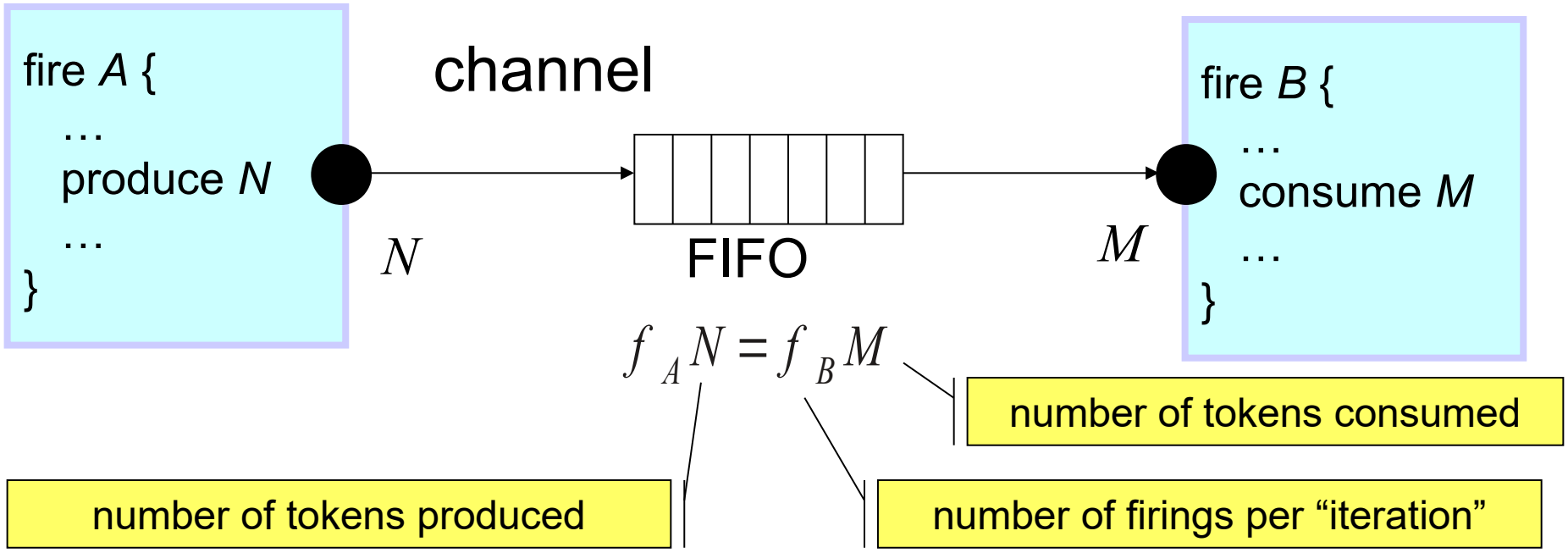
Actual modeling of buffer capacity

The capacity of buffers can be modeled easier:
as a ***backward*** edge where initial number of tokens
= buffer capacity.



Firing rate depends on # of tokens …

Multi-rate models & balance equations (one for each channel)



Decidable:

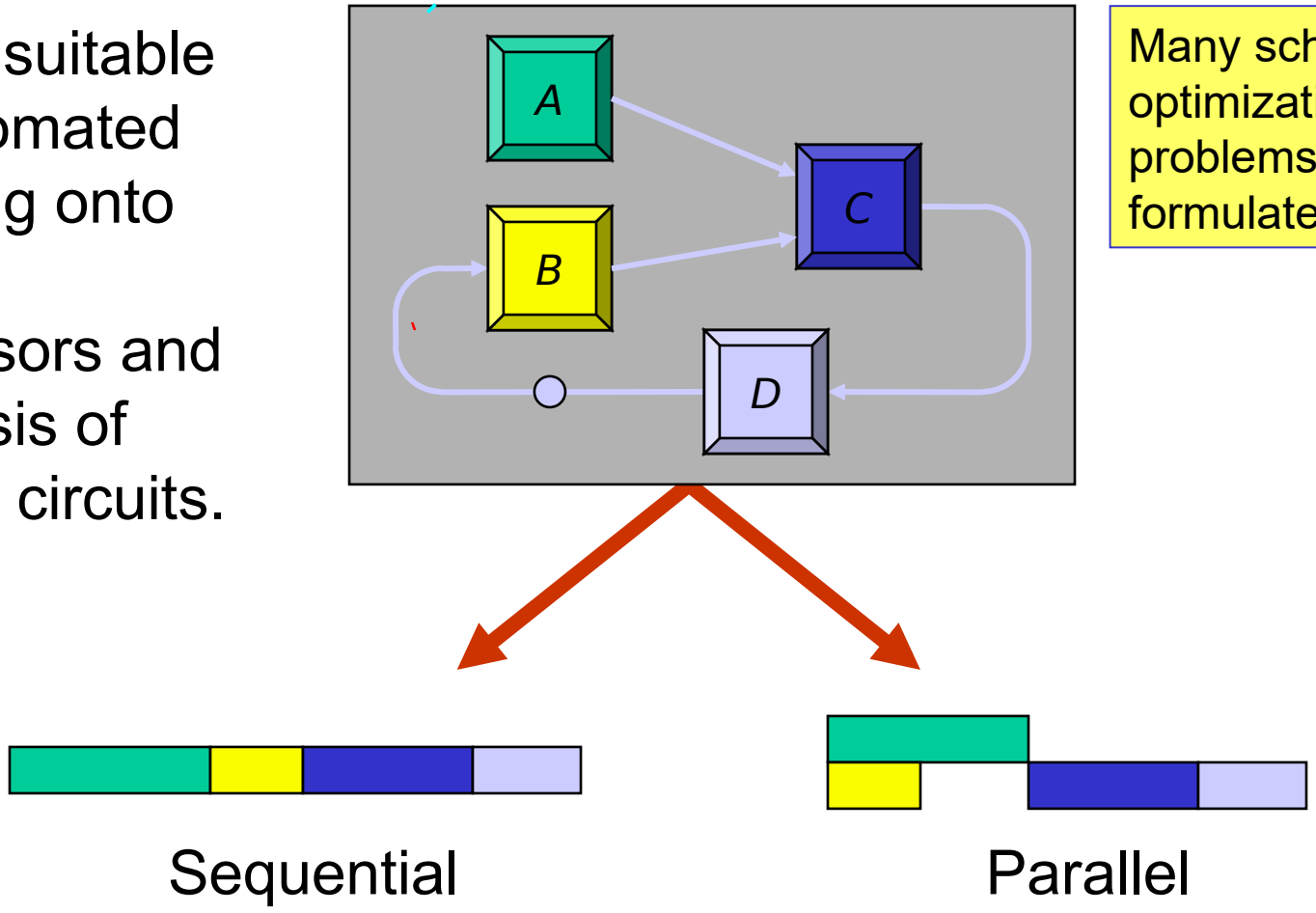
- buffer memory requirements
- deadlock

Schedulable statically

Adopted from: ptolemy.eecs.berkeley.edu/presentations/03/streamingEAL.ppt

Parallel Scheduling of SDF Models

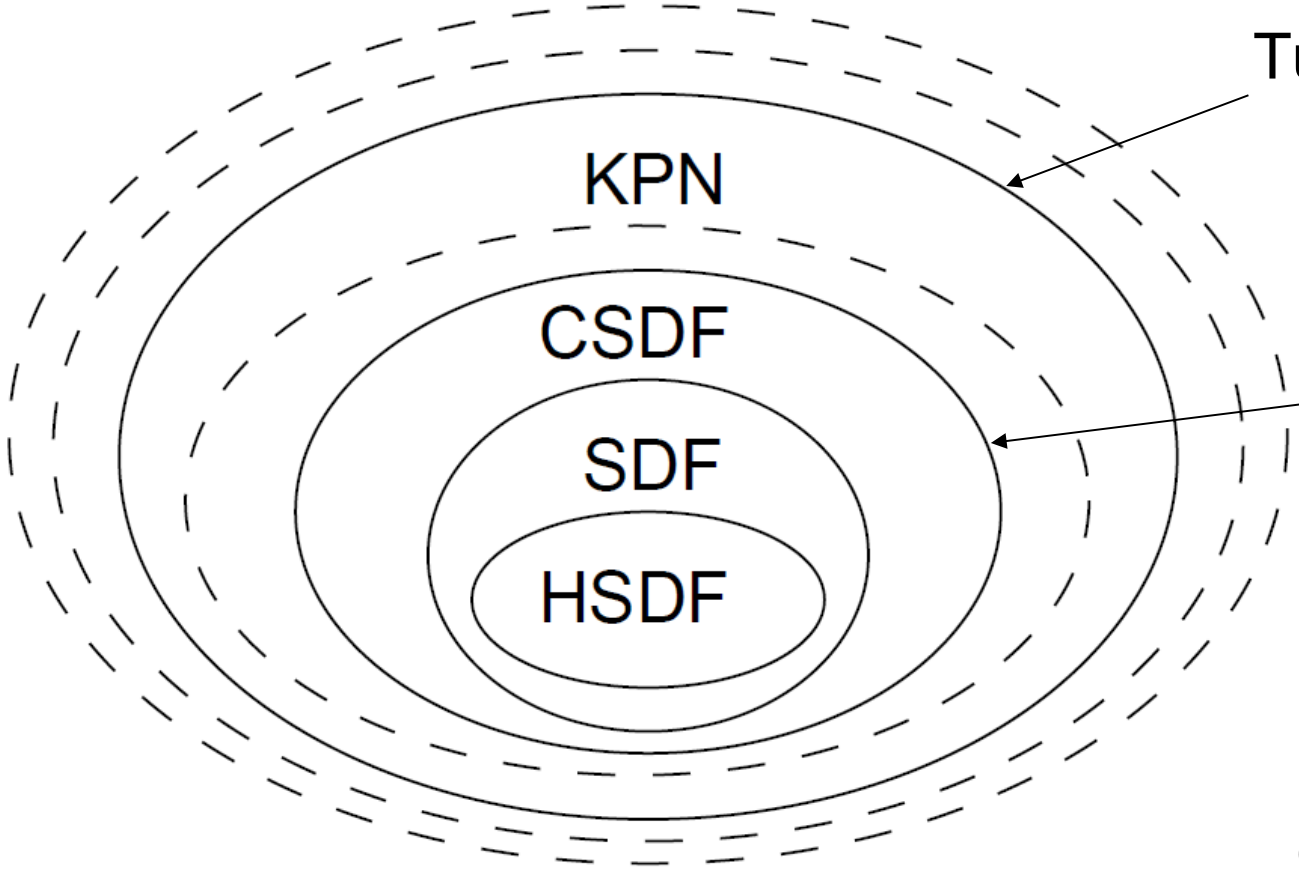
SDF is suitable for automated mapping onto parallel processors and synthesis of parallel circuits.



Many scheduling optimization problems can be formulated.

Source: ptolemy.eecs.berkeley.edu/presentations/03/streamingEAL.ppt

Expressiveness of data flow MoCs



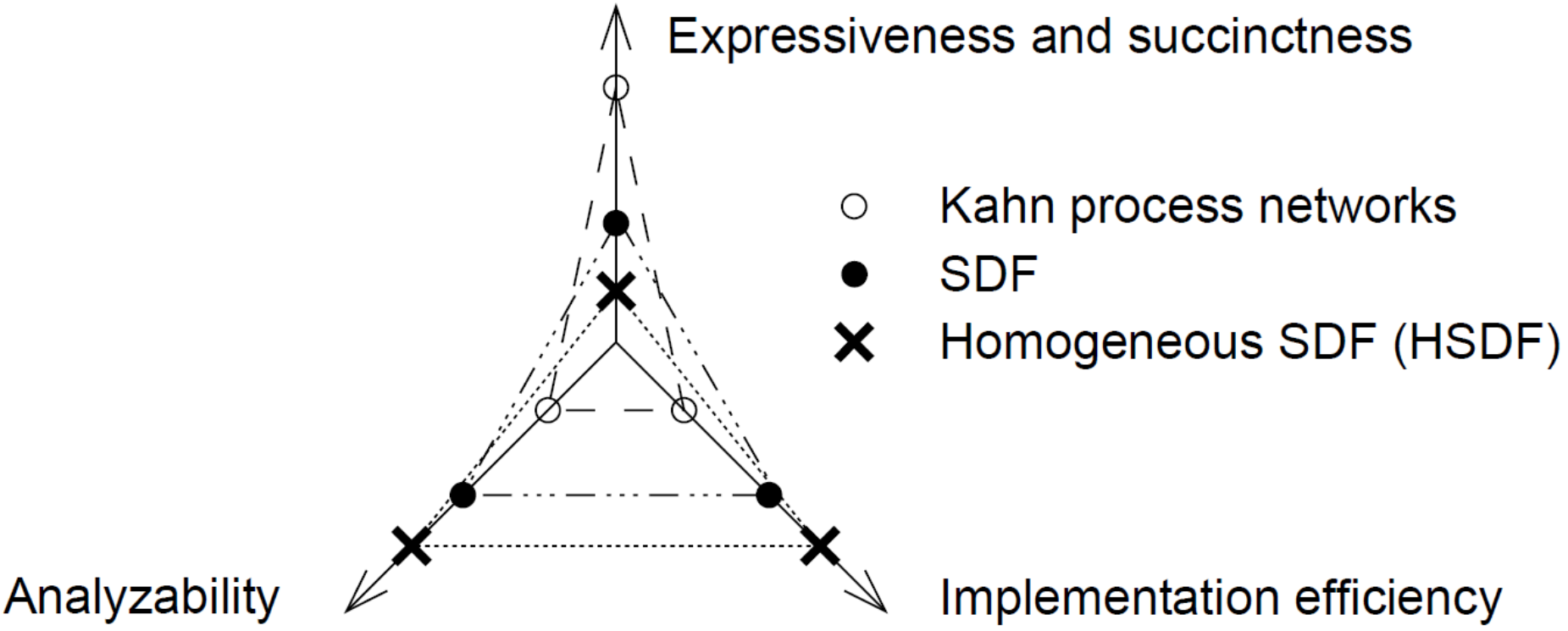
Turing-complete

Not Turing-complete

CSDF=Cyclo static data flow (rates vary in a cyclic way)

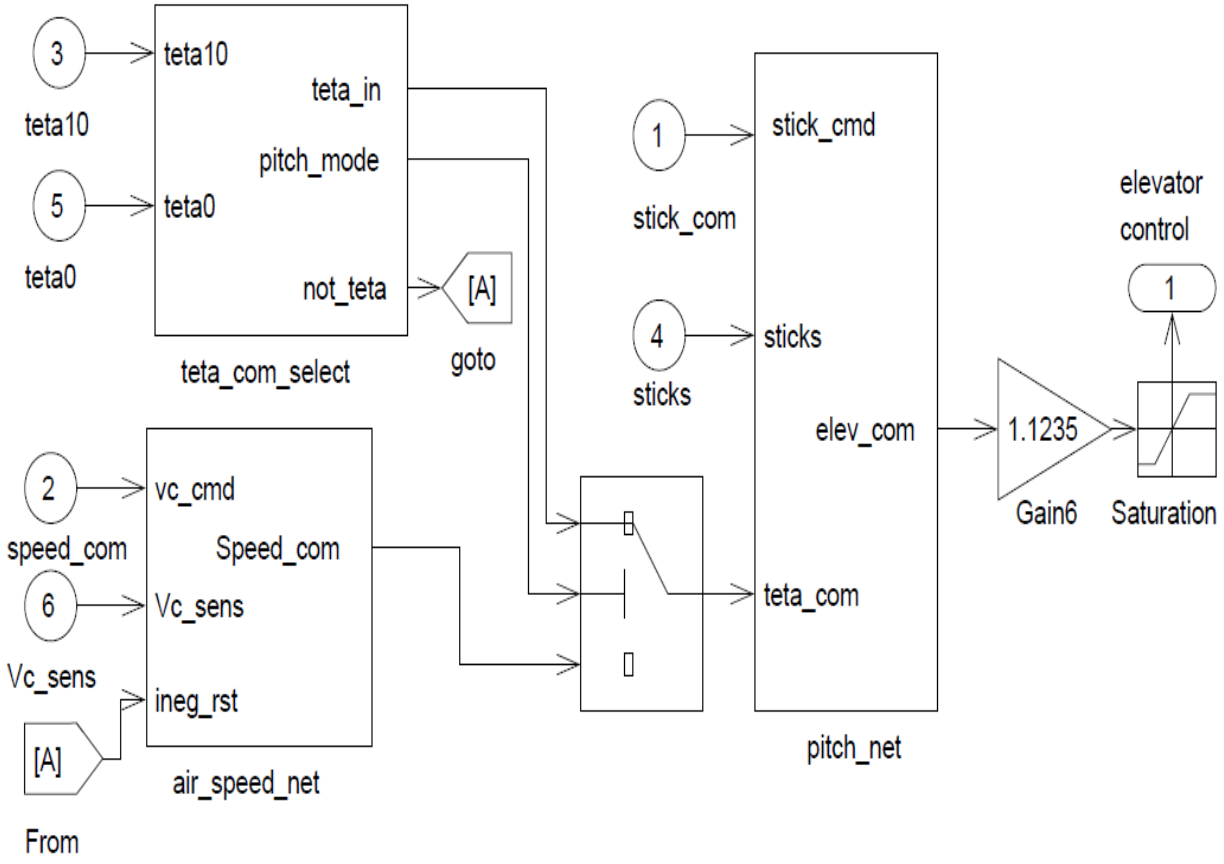
S. Stuijk, 2007

The expressiveness/analyzability conflict



S. Stuijk, 2007

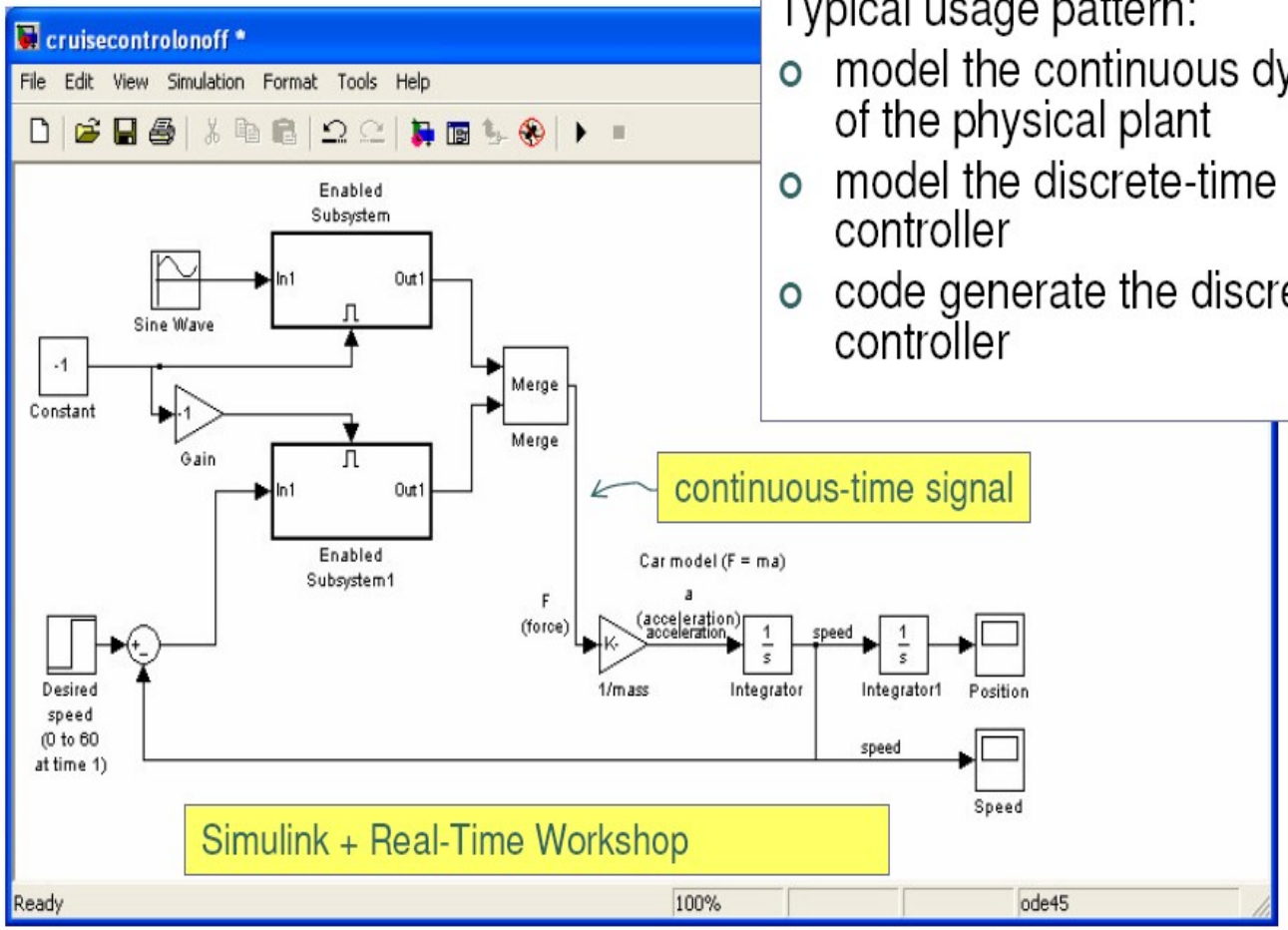
Similar MoC: Simulink
- example -



Semantics? *"Simulink uses an idealized timing model for block execution and communication. Both happen infinitely fast at exact points in simulated time. Thereafter, simulated time is advanced by exact time steps. All values on edges are constant in between time steps."*
[Nicolae Marian, Yue Ma]

de.mathworks.com

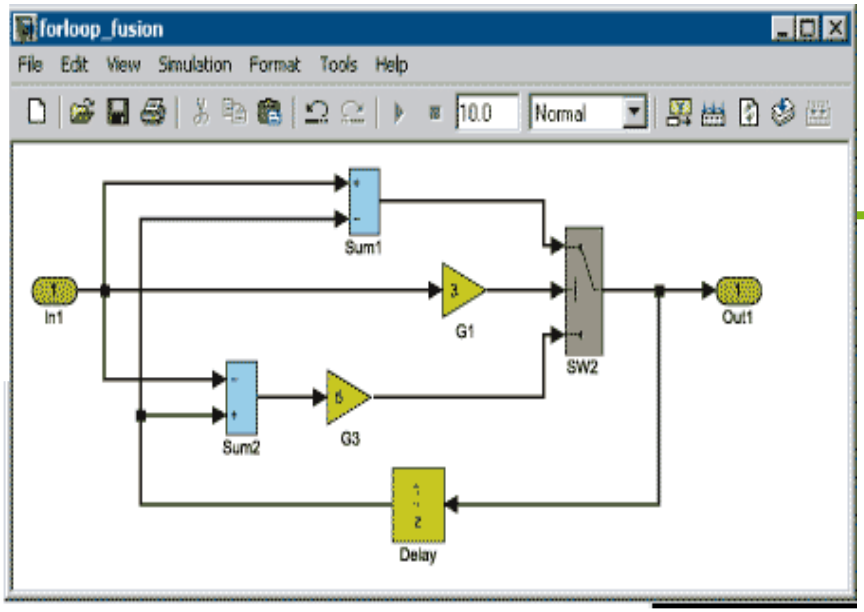
Threads are Not the Only Possibility: 6th example: Continuous-Time Languages



Typical usage pattern:

- model the continuous dynamics of the physical plant
- model the discrete-time controller
- code generate the discrete-time controller

Starting point for “model-based design”



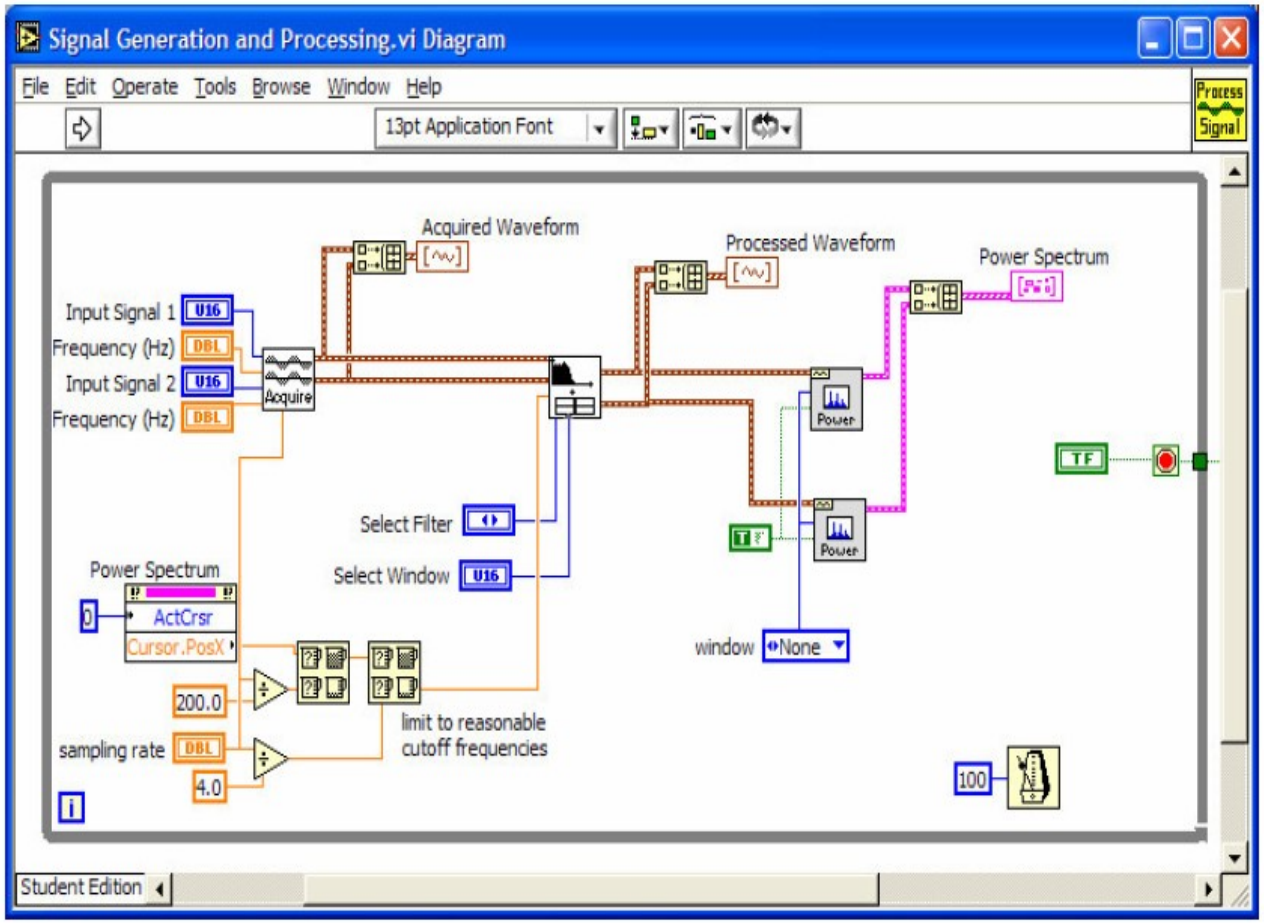

```
/* Switch: '<Root>/SW2' incorporates:
 *  Sum: '<Root>/Sum1'
 *  Gain: '<Root>/G1'
 *  Sum: '<Root>/Sum2'
 *  Gain: '<Root>/G3'
 */
for(i1=0; i1<10; i1++) {
  if(rtU.In1[i1] * 3.0 >= 0.0) {
    rtb_SW2_c[i1] = rtU.In1[i1] - rtDWork.Delay_DSTATE[i1];
  } else {
    rtb_SW2_c[i1] = (rtDWork.Delay_DSTATE[i1] - rtU.In1[i1]) * 5.0;
  }

  /* Outport: '<Root>/Out1' */
  rtY.Out1[i1] = rtb_SW2_c[i1];

  /* Update for UnitDelay: '<Root>/Delay' */
  rtDWork.Delay_DSTATE[i1] = rtb_SW2_c[i1];
}
```

Code automatically generated

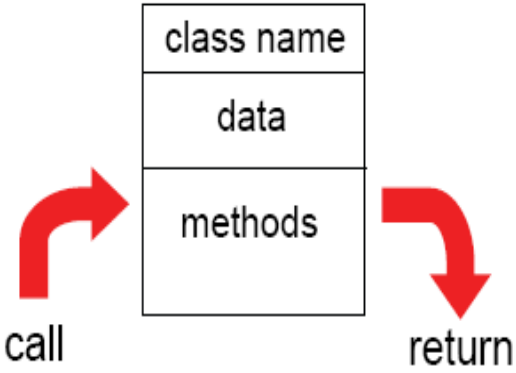
Threads are Not the Only Possibility: 5th example: Instrumentation Languages



e.g. LabVIEW, Structured dataflow model of computation

Actor languages

The established: Object-oriented:

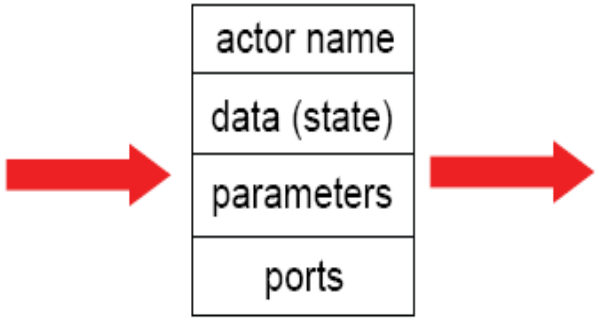


What flows through an object is sequential control

Things happen to objects

The alternative: Actor oriented:

Actors make things happen



What flows through an object is streams of data

Summary

Data flow model of computation

- Motivation, definition
- Kahn process networks (KPNs)
- (H/C)SDF
- Visual programming languages
  - Simulink, Real Time Workshop, LabVIEW

Petri Nets

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 10 月 31 日

Models of computation considered in this course

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorith. ☞ Computer arch.		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC* …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

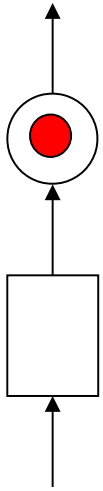
Introduction

Introduced in 1962 by Carl Adam Petri in his PhD thesis.

Focus on modeling causal dependencies;
no global synchronization assumed (message passing only).

Key elements:

- **Conditions**
  Either met or not met.
- **Events**
  May take place if certain conditions are met.
- **Flow relation**
  Relates conditions and events.



Conditions, events and the flow relation form
a **bipartite graph** (graph with two kinds of nodes).

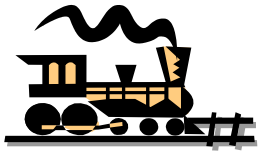
Applications

- modeling of resources
- modeling of mutual exclusion
- modeling of synchronization

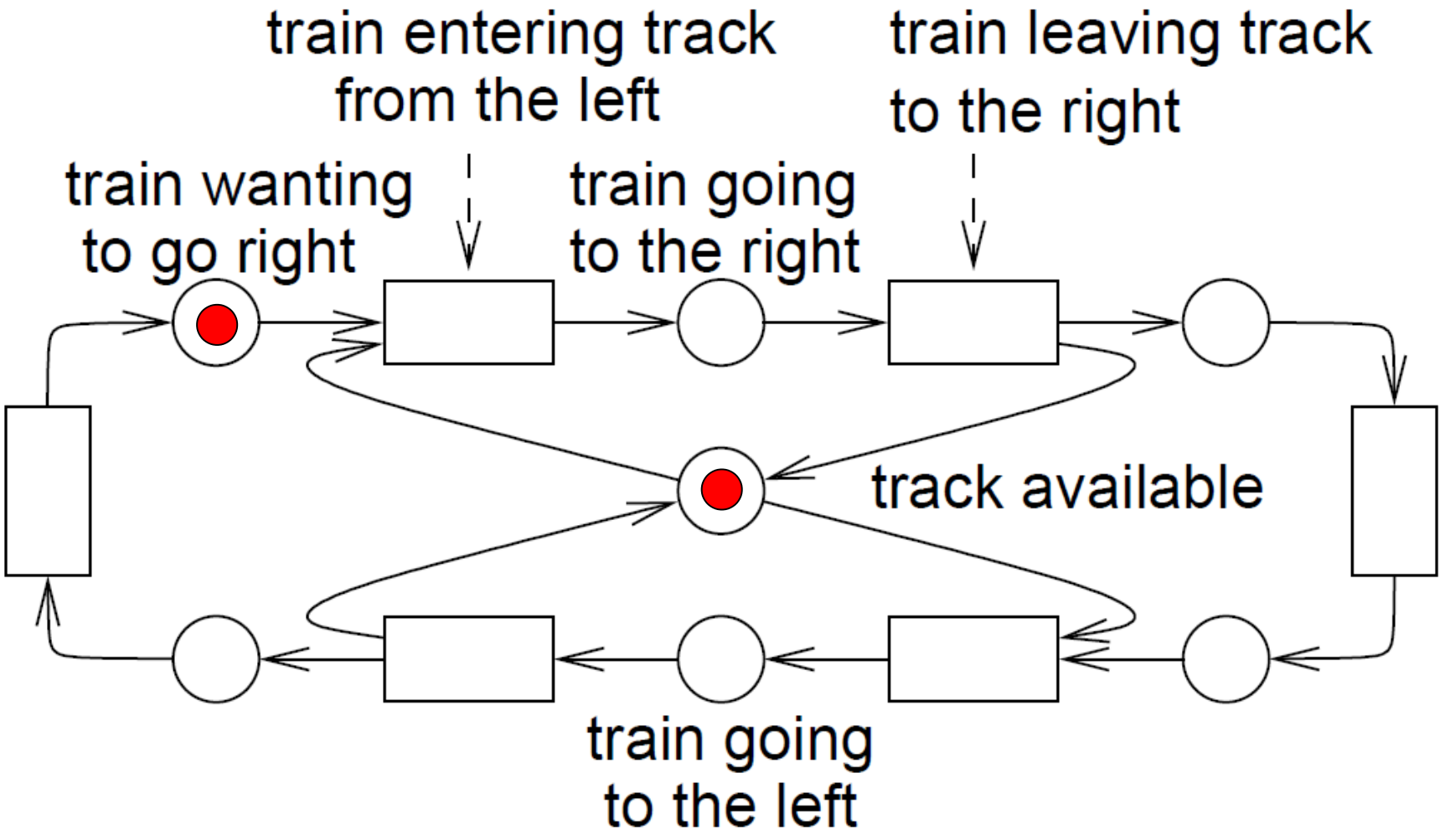
Interactive Tutorials on Petri Nets

http://www.informatik.uni-hamburg.de/TGI/PetriNets/introductions/aalst/

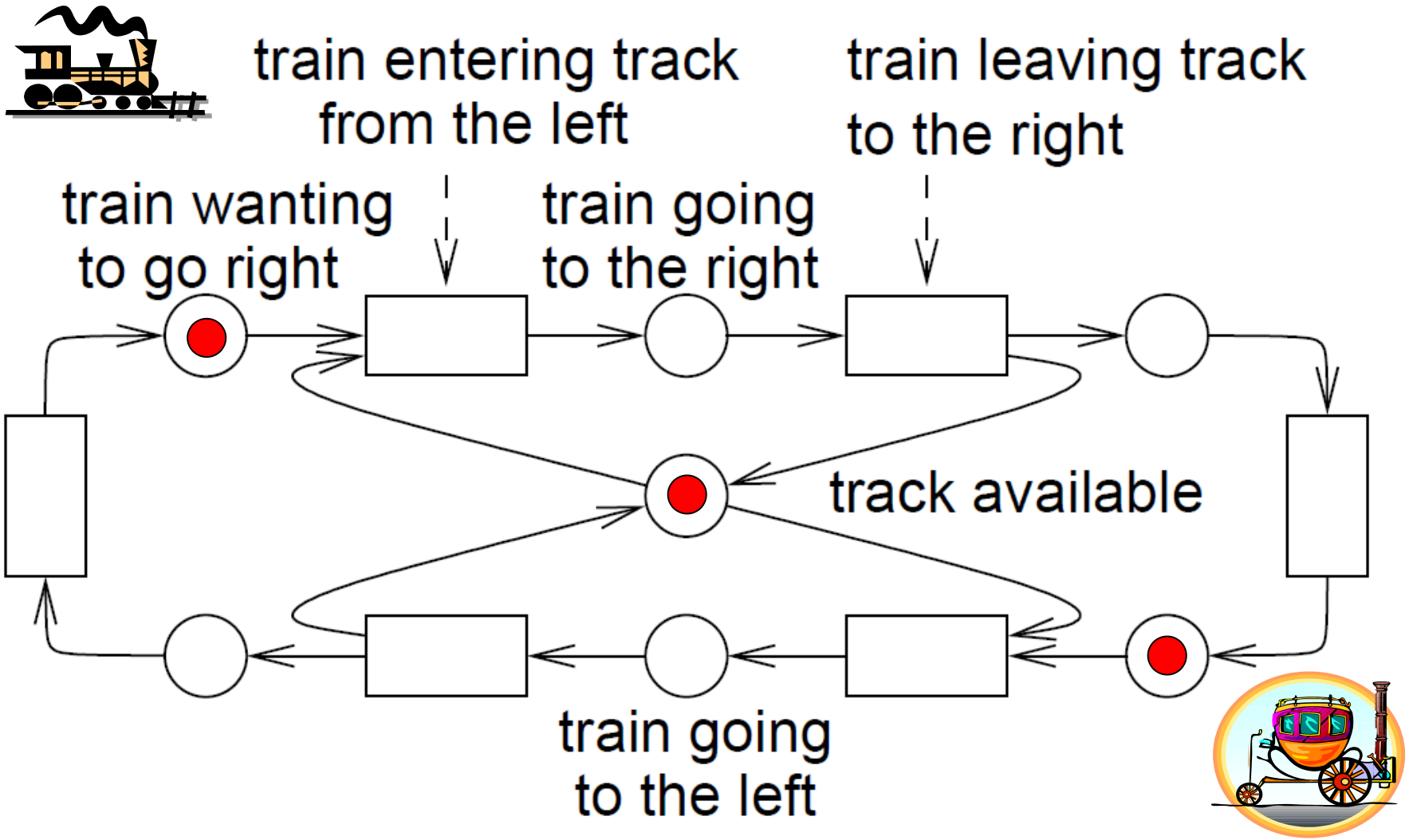
Example: Synchronization at single track rail segment



Playing the “token game“



Conflict for resource “track“

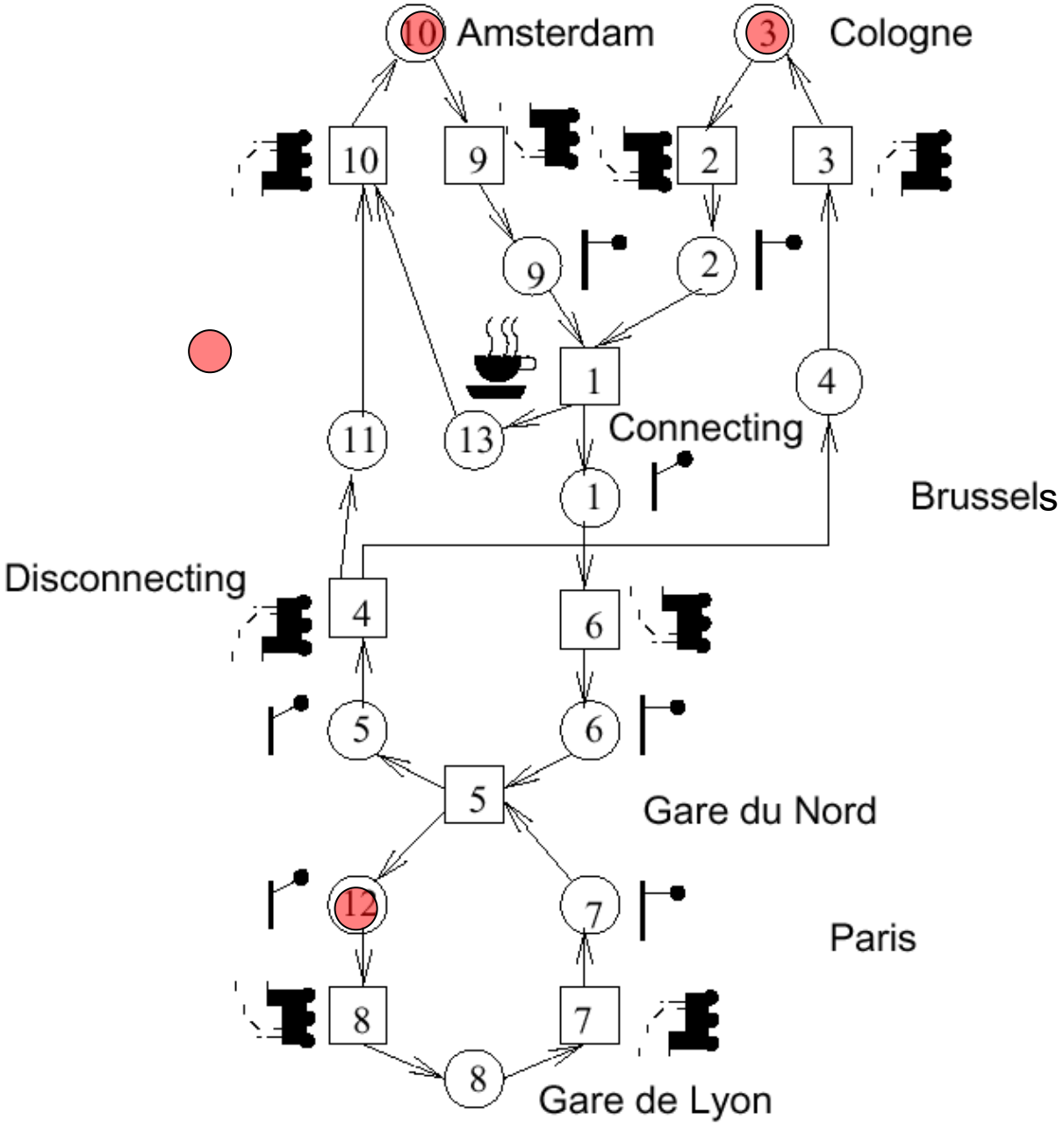


More complex example

Thalys trains between Cologne, Amsterdam, Brussels and Paris.

http://www.thalys.com/be/en

More complex example (2)



Slightly simplified: Synchronization at Brussels and Paris, using stations "Gare du Nord" and "Gare de Lyon" at Paris

Condition/event nets

Def.: $N=(C,E,F)$ is called a **net**, iff the following holds

1. C and E are disjoint sets
2. $F \subseteq (C \times E) \cup (E \times C)$; is binary relation, ("**flow relation**")

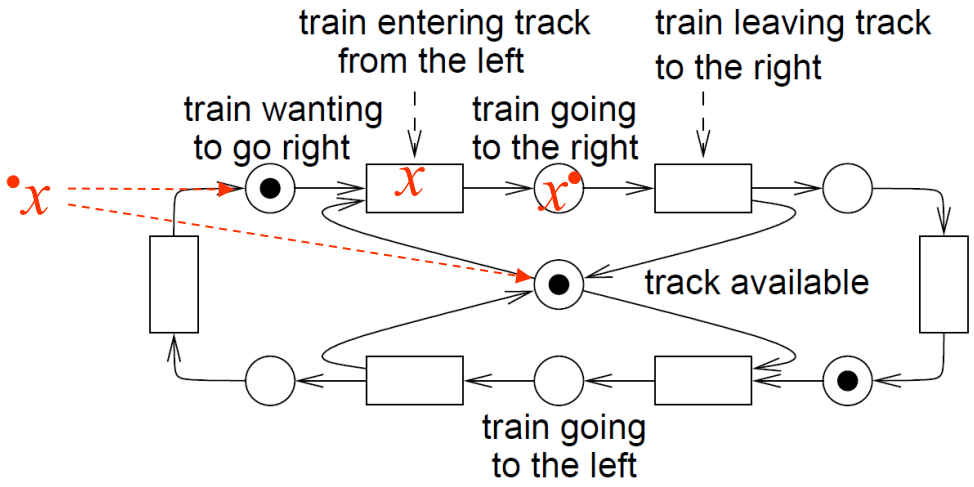
Pre- and post-sets

Def.: Let N be a net and let $x \in (C \cup E)$.

$^\bullet x := \{y \mid y\ F\ x\}$ is called the **pre-set** of x,
(or **preconditions** if $x \in E$)

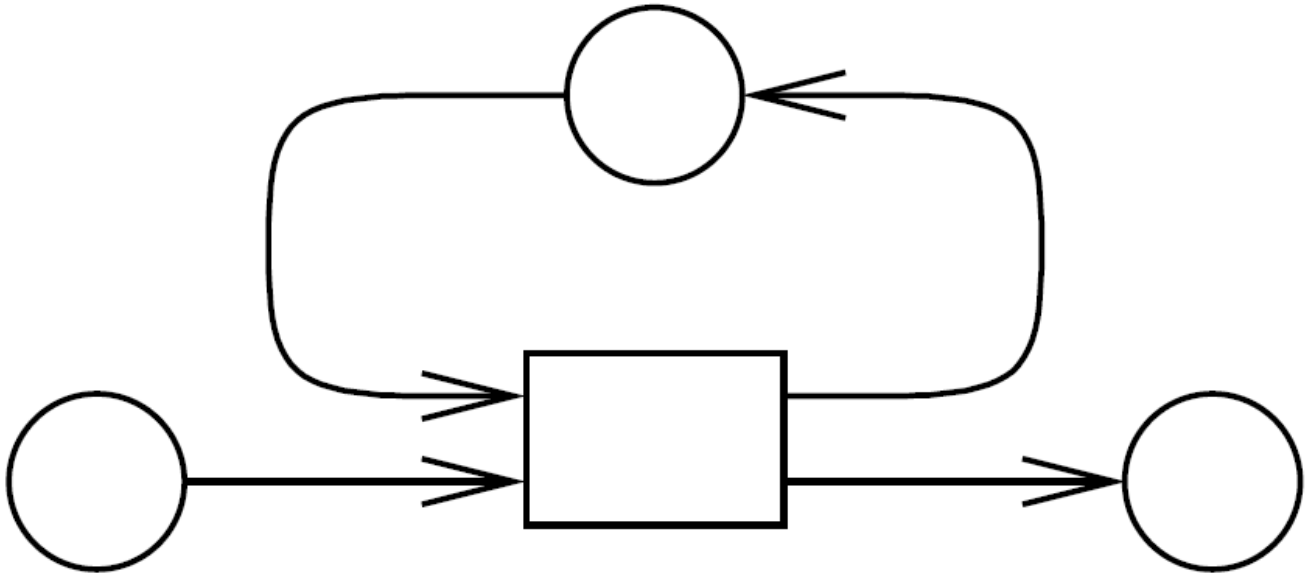
$x^\bullet := \{y \mid x\ F\ y\}$ is called the set of **post-set** of x,
(or **postconditions** if $x \in E$)

Example:



Loops and pure nets

Def.: Let $(c,e) \in C \times E$. (c, e) is called a **loop** iff $cFe \wedge eFc$.

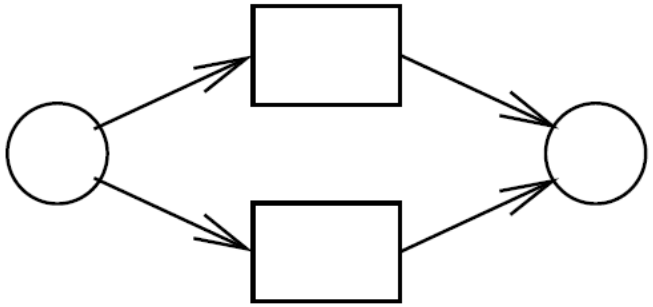


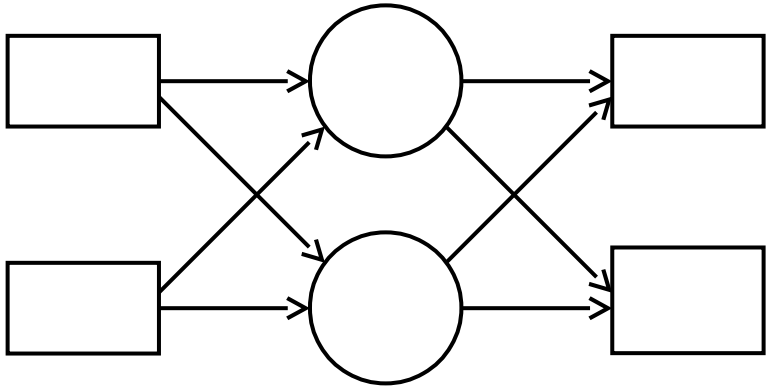
Def.: Net $N=(C,E,F)$ is called **pure**, if F does not contain any loops.

Simple nets

Def.: A net is called **simple** if no two nodes $n1$ and $n2$ have the same pre-set and post-set.

Example (not simple nets):



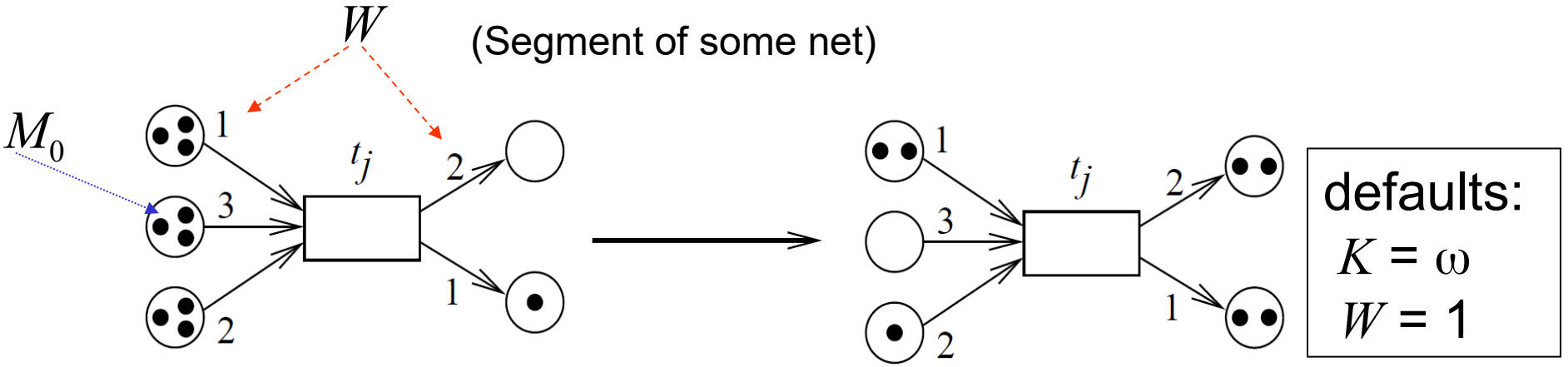


Def.: Simple nets with no isolated elements meeting some additional restrictions are called **condition/event nets (C/E nets)**.

Place/transition nets

Def.: (P, T, F, K, W, M_0) is called a **place/transition net** iff

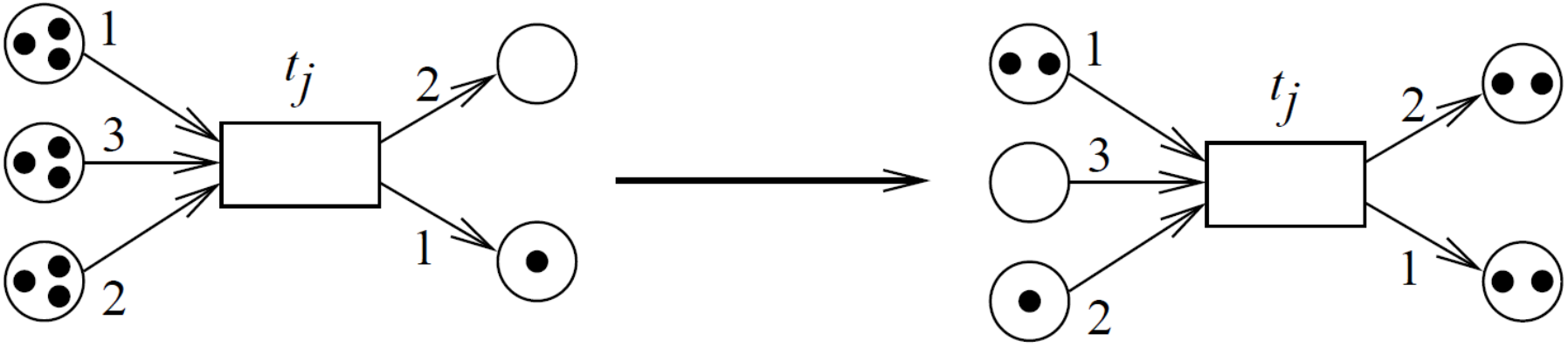
1. $N=(P, T, F)$ is a **net** with places $p \in P$ and transitions $t \in T$
2. K: $P \rightarrow (\mathbb{N}_0 \cup \{\omega\}) \setminus \{0\}$ denotes the **capacity** of places (ω symbolizes infinite capacity)
3. W: $F \rightarrow (\mathbb{N}_0 \setminus \{0\})$ denotes the **weight of graph edges**
4. M_0: $P \rightarrow \mathbb{N}_0 \cup \{\omega\}$ represents the **initial marking** of places



Computing changes of markings

"Firing" transitions t generate new markings on each of the places p according to the following rules:

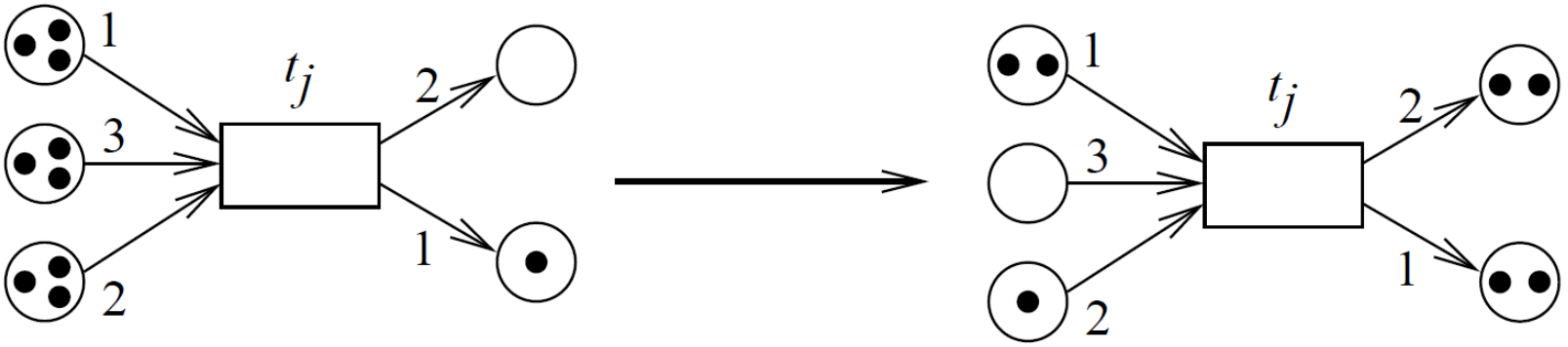
$$M'(p) = \begin{cases} M(p) - W(p,t), & \text{if } p \in {}^\bullet t \setminus t^\bullet \\ M(p) + W(p,t), & \text{if } p \in t^\bullet \setminus {}^\bullet t \\ M(p) - W(p,t) + W(p,t), & \text{if } p \in {}^\bullet t \cap t^\bullet \\ M(p) & \text{otherwise} \end{cases}$$



Activated transitions

Transition t is “activated“ iff

$$(\forall\, p \in {}^\bullet t:\ M(p) \geq W(p,t)) \ \wedge\ (\forall\, p \in t^\bullet:\ M(p) + W(t,p) \leq K(p))$$



- Activated transitions can “take place“ or “fire“, but don‘t have to.
- We never talk about “time“ in the context of Petri nets.
- The order in which activated transitions fire, is not fixed (it is non-deterministic).

Shorthand for changes of markings

Slide 15:

$$M'(p) = \begin{cases} M(p) - W(p,t), & \text{if } p \in {}^\bullet t \setminus t^\bullet \\ M(p) + W(p,t), & \text{if } p \in t^\bullet \setminus {}^\bullet t \\ M(p) - W(p,t) + W(p,t), & \text{if } p \in t^\bullet \cap {}^\bullet t \\ M(p) & \text{otherwise} \end{cases}$$

Let

$$\underline{t}(p) = \begin{cases} -W(p,t), & \text{if } p \in {}^\bullet t \setminus t^\bullet \\ +W(p,t), & \text{if } p \in t^\bullet \setminus {}^\bullet t \\ -W(p,t) + W(p,t), & \text{if } p \in t^\bullet \cap {}^\bullet t \\ 0 & \text{otherwise} \end{cases}$$

$$\Rightarrow \quad \forall p \in P: M'(p) = M(p) + \underline{t}(p)$$

$$\Rightarrow \quad M' = M + \underline{t} \qquad +: \text{vector add}$$

Matrix $\underline{N}$ describing all changes of markings

$$\underline{t}(p) = \begin{cases} -W(p,t), & \text{if } p \in {}^\bullet t \setminus t^\bullet \\ +W(p,t), & \text{if } p \in t^\bullet \setminus {}^\bullet t \\ -W(p,t)+W(p,t), & \text{if } p \in t^\bullet \cap {}^\bullet t \\ 0 & \text{otherwise} \end{cases}$$

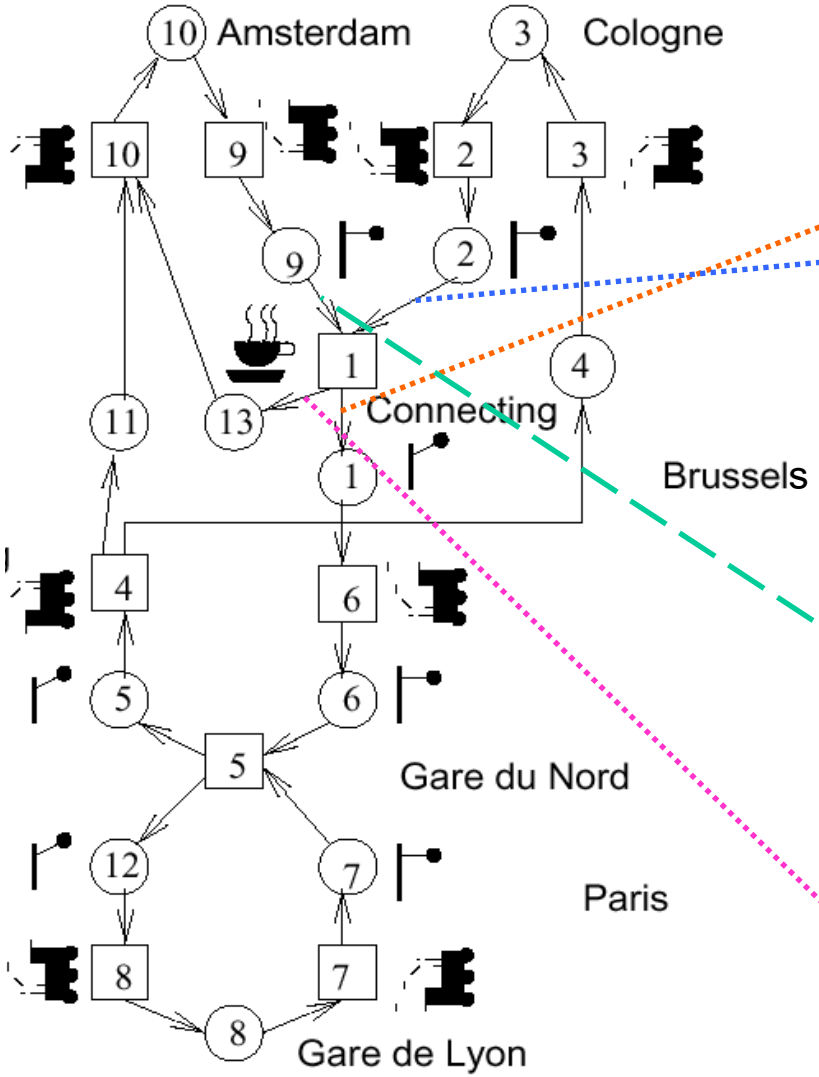
Def.: Matrix $\underline{N}$ of net N is a mapping

$$\underline{N}: P \times T \rightarrow \mathbb{Z} \text{ (integers)}$$

such that $\forall\, t \in T$: $\underline{N}(p,t) = \underline{t}(p)$

- Component in column t and row p indicates the change of the marking of place p if transition t takes place.
- For pure nets, $(\underline{N}, M_0)$ is a complete representation of a net.

Example: N =



	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
p_1	1					−1				
p_2	−1	1								
p_3		−1	1							
p_4			−1	1						
p_5				−1	1					
p_6					−1	1				
p_7					−1		1			
p_8							−1	1		
p_9	−1								1	
p_{10}									−1	1
p_{11}				1						−1
p_{12}					1			−1		
p_{13}	1									−1

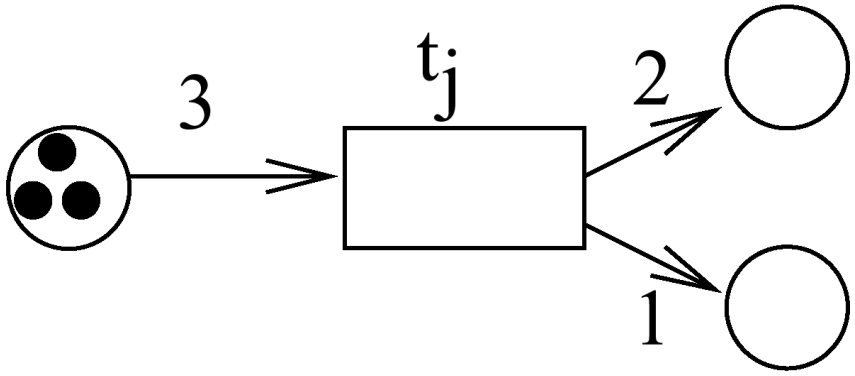
Place - invariants

Standardized technique for proving properties of system models

For any transition $t_j \in T$ we are looking for sets $R \subseteq P$ of places for which the accumulated marking is constant:

$$\sum_{p \in R} \underline{t}_j(p) = 0$$

Example:



Predicate/transition nets

Goal: compact representation of complex systems.

Key changes:

- Tokens are becoming individuals;
- Transitions enabled if functions at incoming edges true;
- Individuals generated by firing transitions defined through functions

Changes can be explained by folding and unfolding C/E nets,

☞ semantics can be defined by C/E nets.

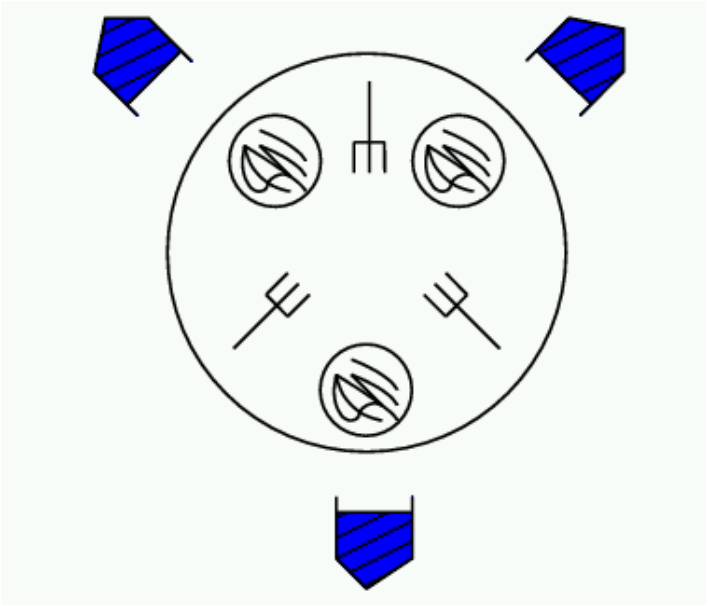
Example: Dining philosophers problem

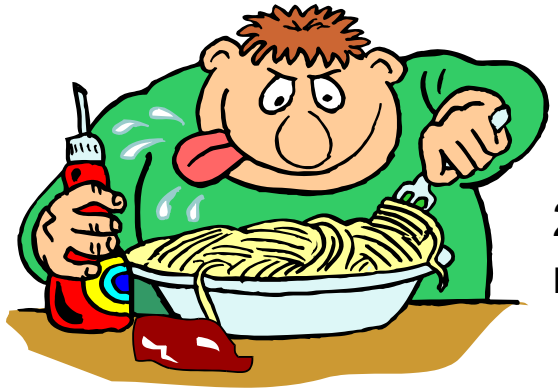
n>1 philosophers sitting at a round table;

n forks,

n plates with spaghetti;

philosophers either thinking or eating spaghetti (using left and right fork).



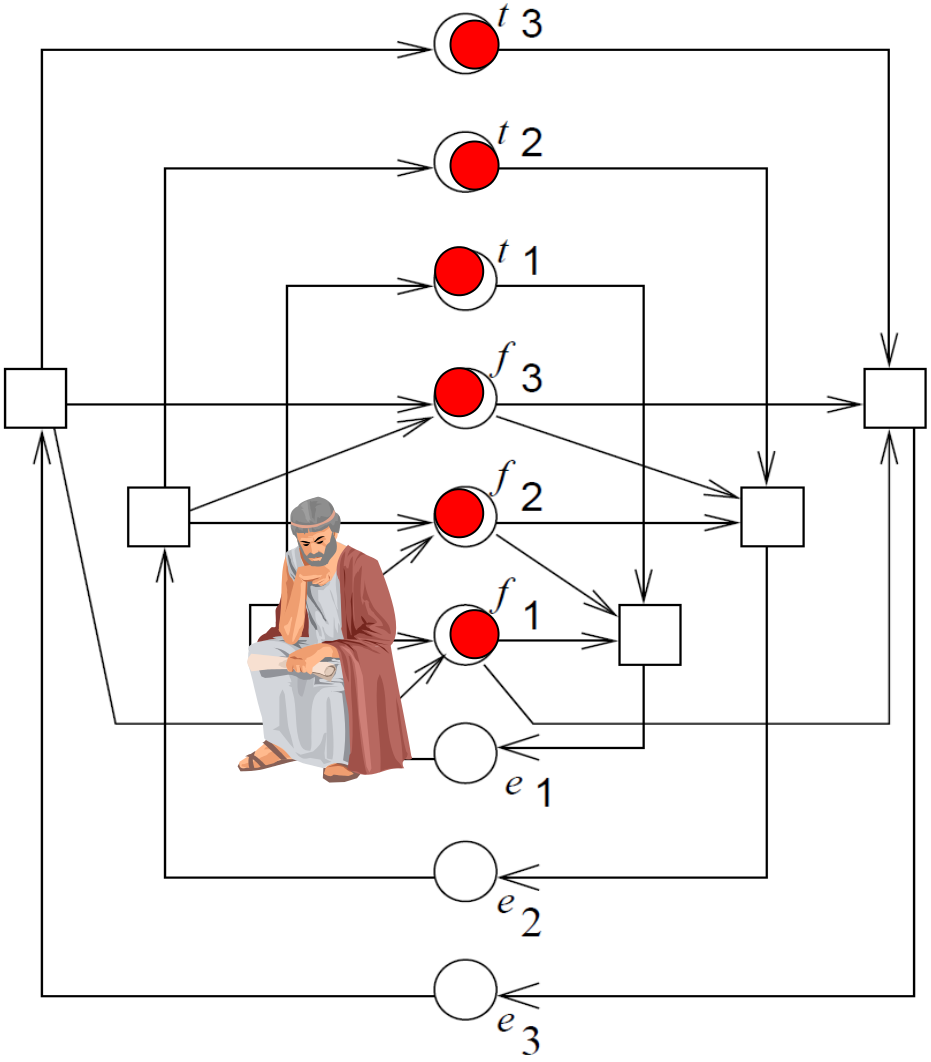


2 forks needed!

How to model conflict for forks?

How to guarantee avoiding starvation?

Condition/event net model of the dining philosophers problem



Let $x \in \{1..3\}$

t_x : x is thinking

e_x : x is eating

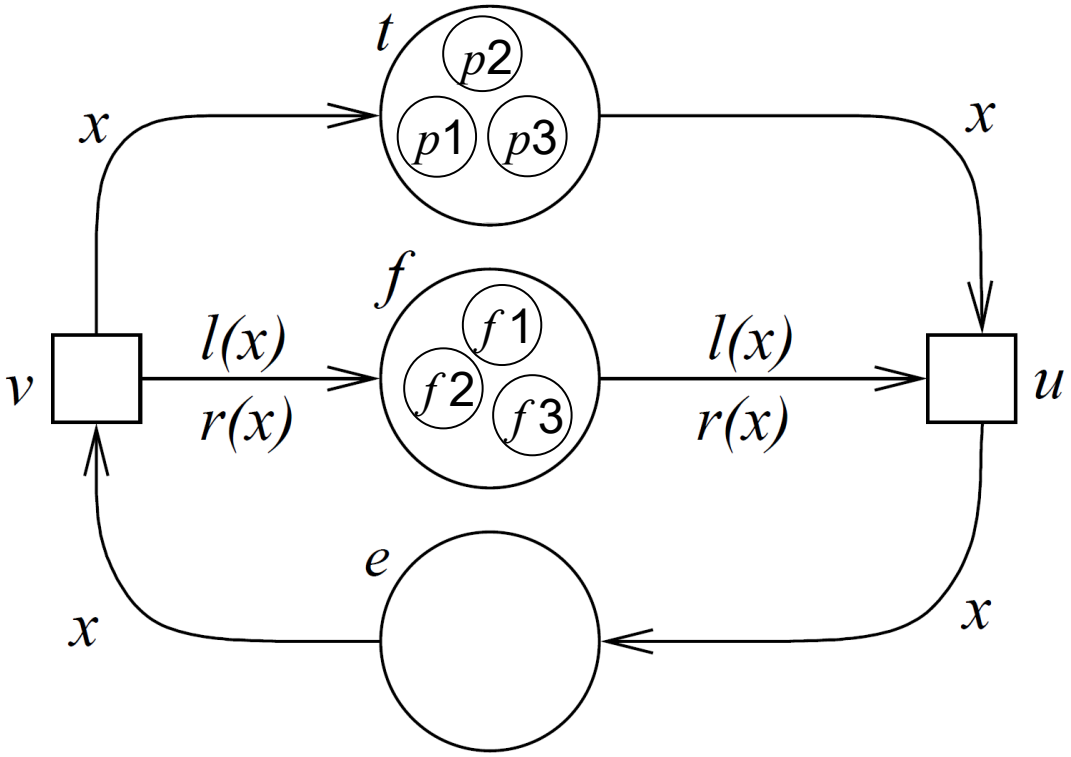
f_x : fork x is available

Model quite clumsy.

Difficult to extend to more philosophers.

Predicate/transition model of the dining philosophers problem (1)

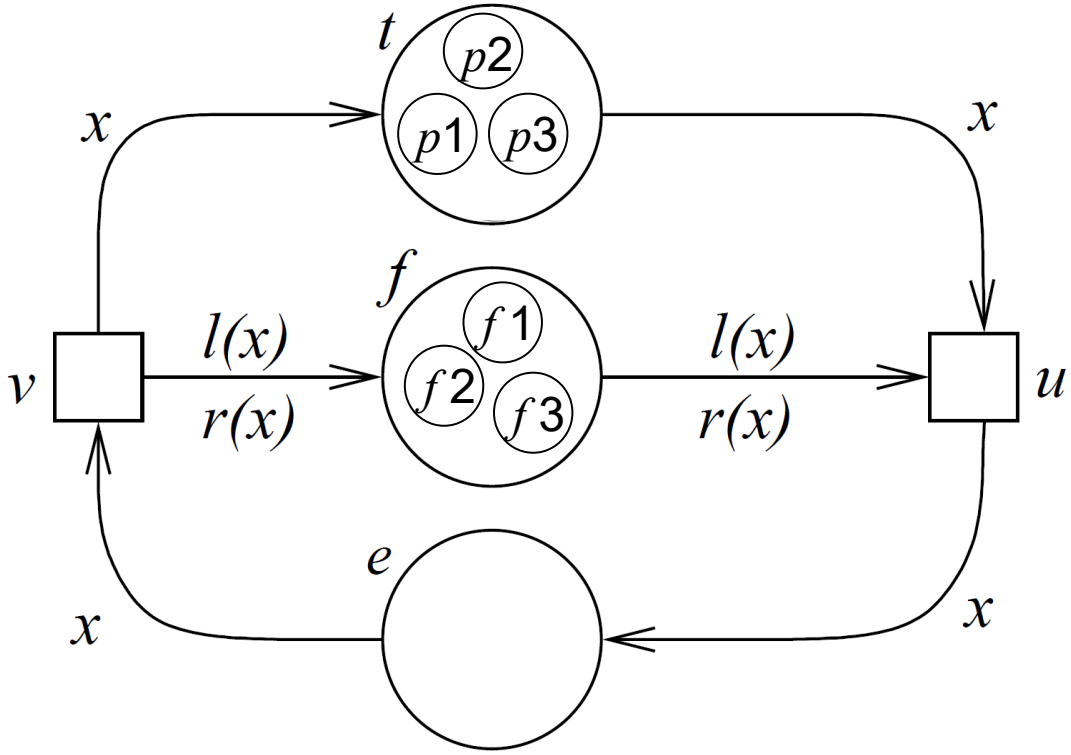
Let x be one of the philosophers,
Let $l(x)$ be the left spoon of x,
Let $r(x)$ be the right spoon of x.



Tokens: individuals.

Semantics can be defined by replacing net by equivalent condition/event net.

Predicate/transition model of the dining philosophers problem (2)



Model can be extended to arbitrary numbers of people.

Evaluation

PROs:

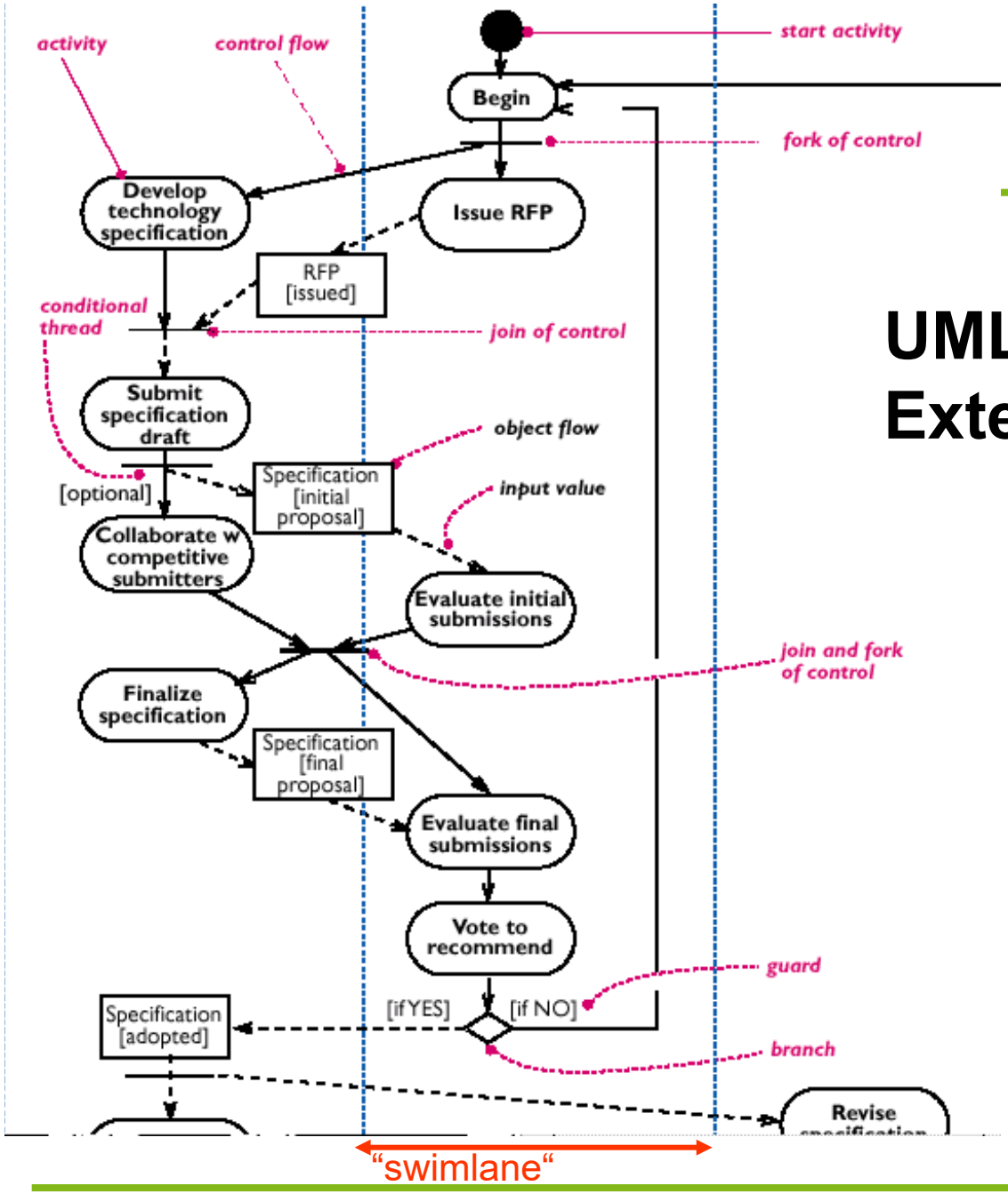
- Appropriate for distributed applications,
- Well-known theory for formally proving properties,
- Initially a quite bizarre topic, but now accepted due to increasing number of distributed applications.

CONs (for the nets presented) **:**

- problems with modeling timing,
- no programming elements,
- no hierarchy.

Extensions:

- Enormous amounts of efforts on removing limitations.



UML Activity diagrams: Extended Petri nets

Include decisions (like in flow charts). Graphical notation similar to SDL.

© Cris Kobryn: UML 2001: A Standardization Odyssey, CACM, October, 1999

Summary

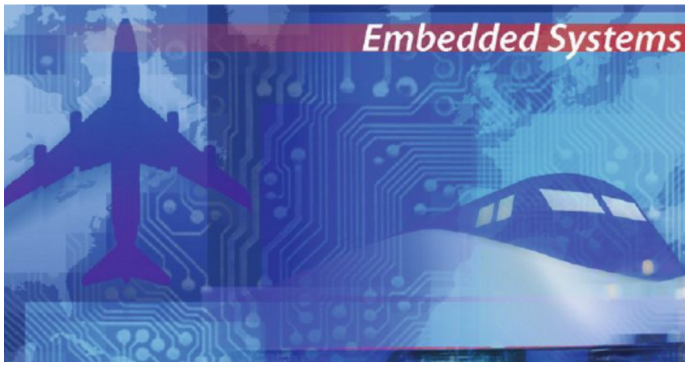
Petri nets: focus on causal dependencies

- Condition/event nets
  - Single token per place
- Place/transition nets
  - Multiple tokens per place
- Predicate/transition nets
  - Tokens become individuals
  - Dining philosophers used as an example
- Extensions required to get around limitations

Activity diagrams in UML are extended Petri nets

Discrete Event Models

Peter Marwedel
TU Dortmund,
Informatik 12



2012 年 11 月 06 日

Models of computation considered in this course

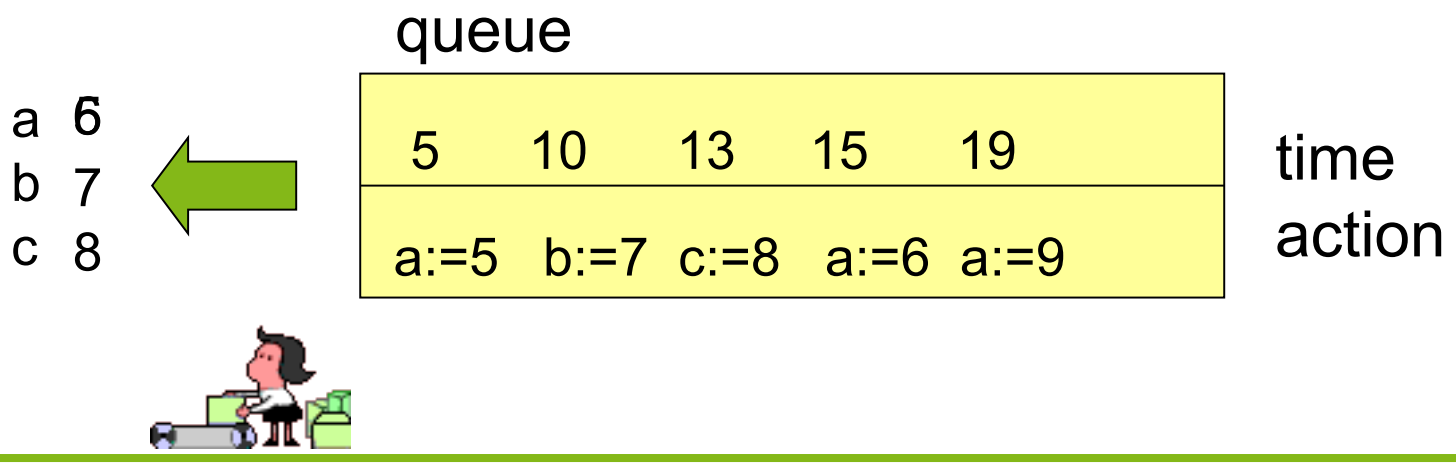
Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorith. ☞ Computer arch.		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC* …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

Discrete event semantics

Basic discrete event (DE) semantics

- Queue of future actions, sorted by time
- Loop:
  - Fetch next entry from queue
  - Perform function as listed in entry
    - May include generation of new entries
- Until termination criterion = true



HDLs using discrete event (DE) semantics

Used in hardware description languages (HDLs):

Description of concurrency is a must for HW description languages!

- Many HW components are operating concurrently
- Typically mapped to “processes“
- These processes communicate via “signals“
- Examples:
  - MIMOLA [Zimmermann/Marwedel], ~1975 …
  - ….
  - VHDL (very prominent example in DE modeling)
    One of the 3 most important HDLs: VHDL, Verilog, SystemC

VHDL

VHDL = VHSIC hardware description language

VHSIC = very high speed integrated circuit

1980: Def. started by US Dept. of Defense (DoD) in 1980

1984: first version of the language defined, based on ADA (which in turn is based on PASCAL)

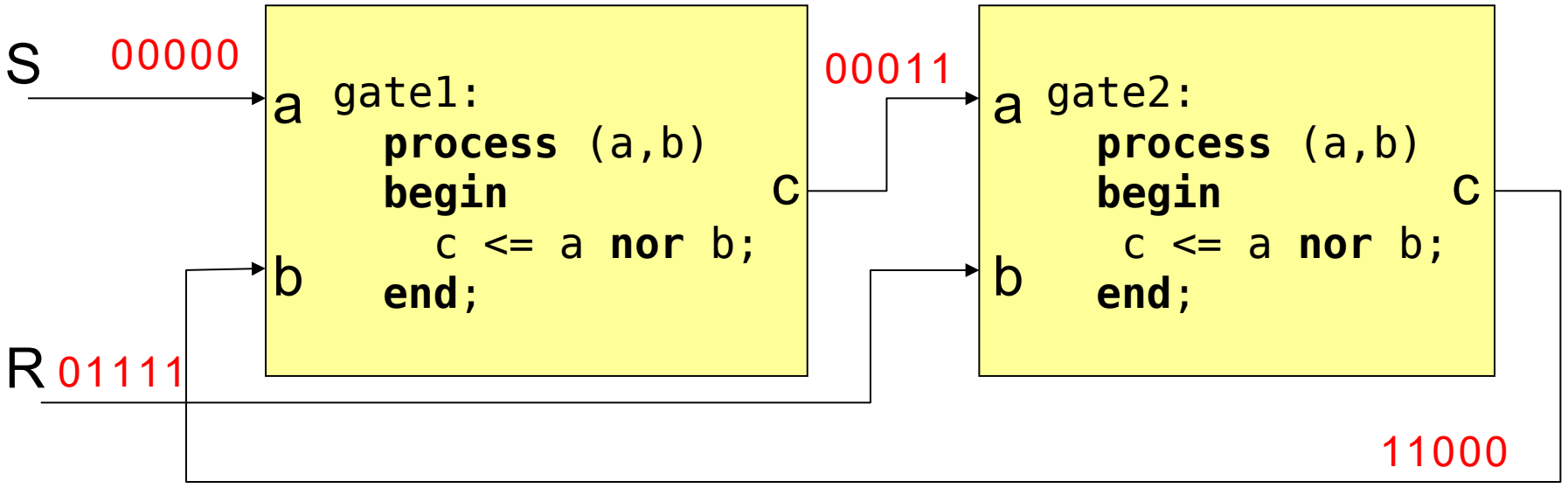
1987: revised version became IEEE standard 1076

1992: revised IEEE standard

1999: VHDL-AMS: includes analog modeling

2006: Major extensions

Simple example (VHDL notation)



- Processes will wait for changes on their input ports
  = process triggered by an event
- If they arrive, processes will wake up, compute their code, deposit changes of output signals in the event queue and wait for the next event
- If all processes wait, the next entry will be taken from the event queue
- This event may trigger several processes (start next simulation cycle)

VHDL processes

Delays allowed:

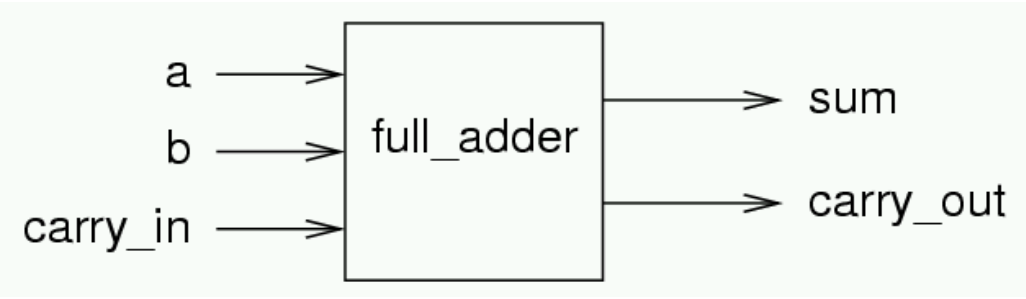
```
process (a,b)
begin
  c <= a nor b after 10 ns;
end;
```

Equivalent to

```
process
begin
  c <= a nor b after 10 ns;
  wait on a,b;
end;
```

- <=: signal assignment operator
- Each executed signal assignment will result in **adding** entries in the projected waveform, as indicated by the (optional) delay time
- Implicit loop around the code in the body
- Sensitivity lists are a shorthand for a single **wait on**-statement at the end of the process body

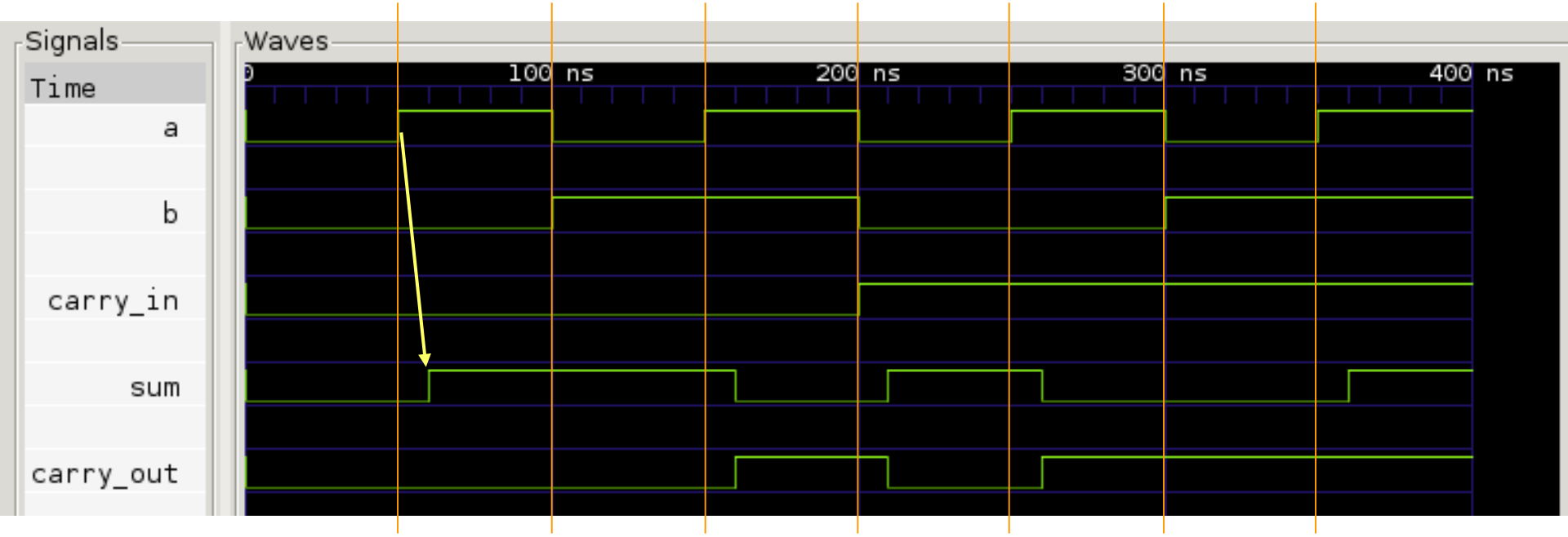
The full adder as an example




```
entity full_adder is
port(a, b, carry_in: in bit;     -- input ports
     sum,carry_out: out bit);    --output ports
end entity full_adder;


architecture behavior of full_adder is
begin
  sum       <= (a xor b) xor carry_in after 10 ns;
  carry_out <= (a and b) or (a and carry_in) or
               (b and carry_in)       after 10 ns;
end architecture behavior;
```

The full adder as an example
- Simulation results -



VHDL semantics: global control

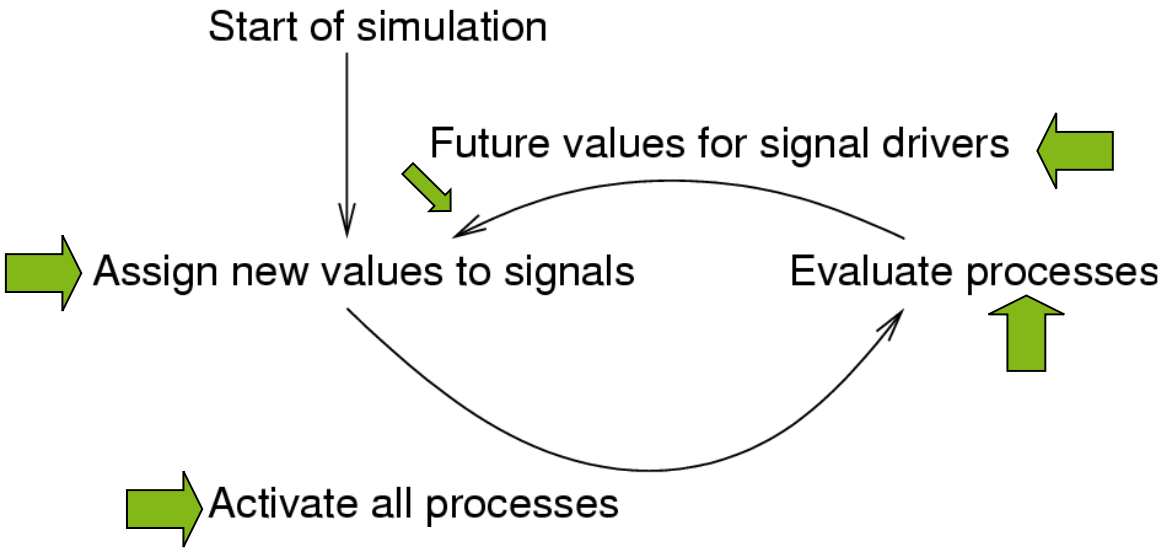
According to the original VHDL standards document:

- The execution of a model consists of an initialization phase followed by the repetitive execution of process statements in the description of that model.
- Initialization phase executes each process once.

VHDL semantics: initialization

At the beginning of initialization, the current time, T_c is 0 ns.

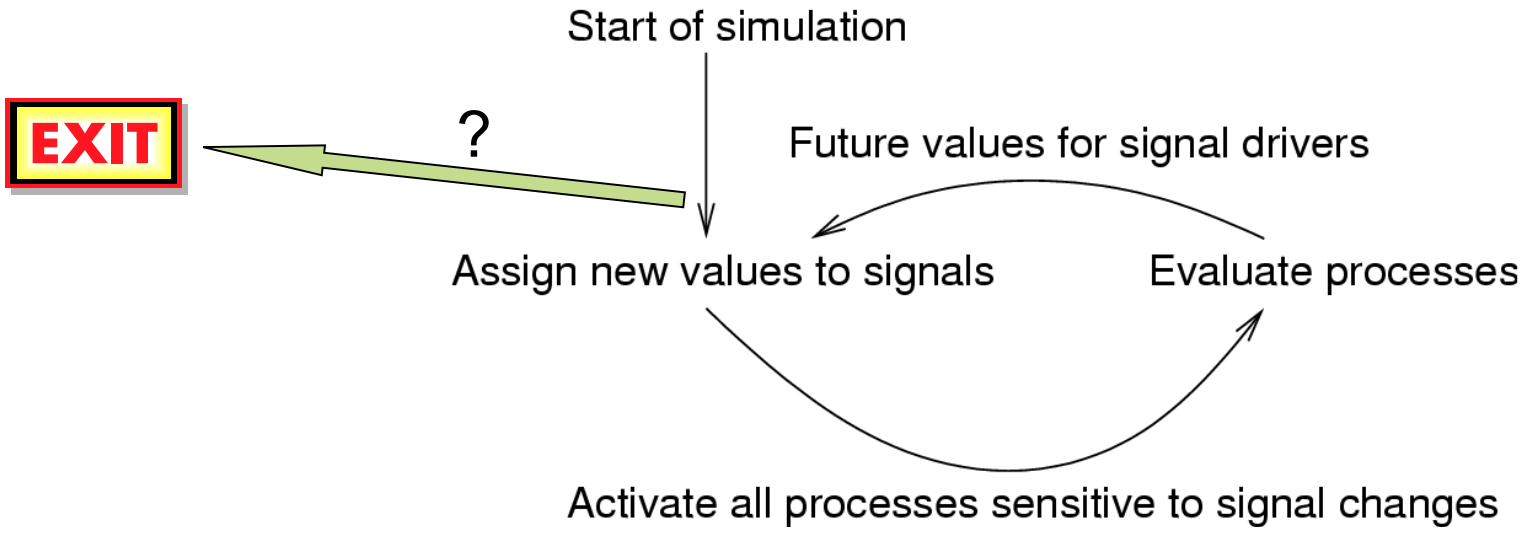
- *The … effective value of each explicitly declared signal are computed, and the current value of the signal is set to the effective value. …*
- *Each ... process … is executed until it suspends*.
- *The time of the next simulation cycle (… in this case … the 1st cycle), T_n is calculated according to the rules of step f of the simulation cycle, below*.



VHDL semantics: The simulation cycle (1)

According to the standard, the simulation cycle is as follows:

a) Stop if T_n= time‘high
and “nothing else is to be done” at T_n.
The current time, T_c is set to T_n.



VHDL semantics: The simulation cycle (2)

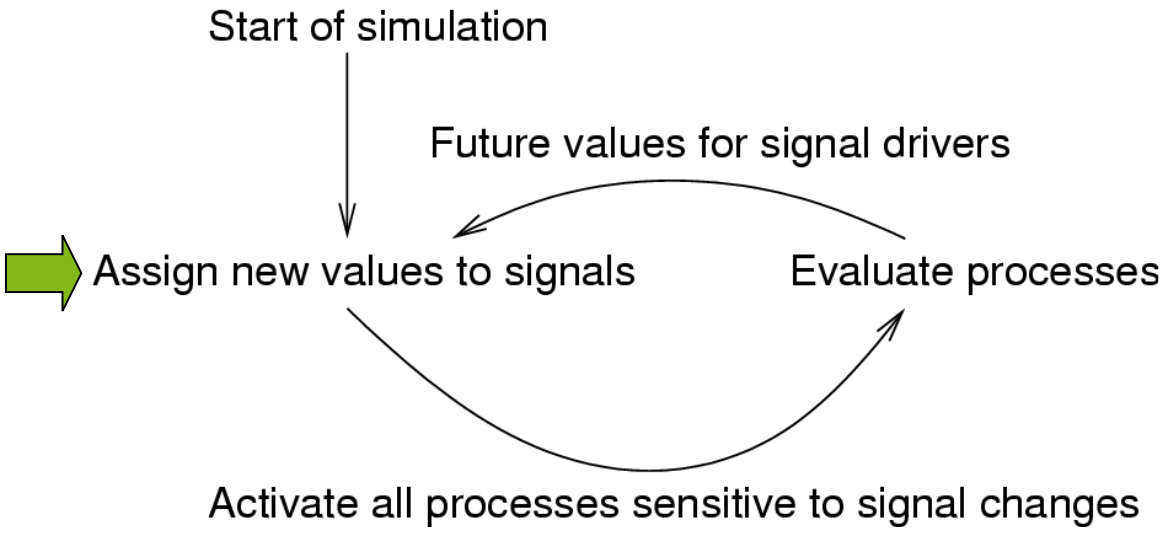
b) Each active explicit signal in the model is updated. (Events may occur as a result.)

Previously computed entries in the queue are now assigned if their time corresponds to the current time T_c.

New values of signals are not assigned before the next simulation cycle, at the earliest.

Signal value changes result in events ☞ enable the execution of processes that are sensitive to that signal.

c) ...

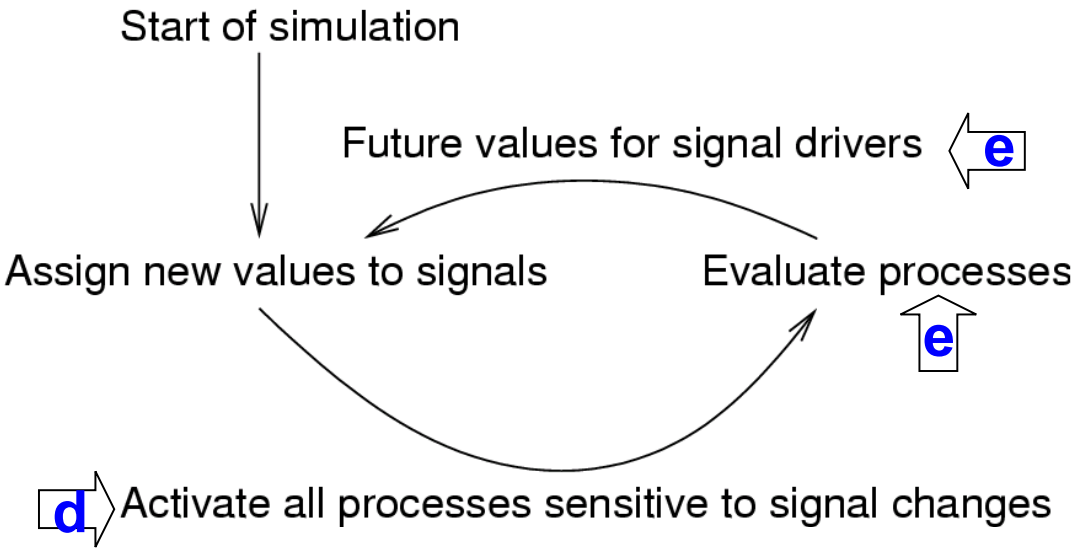


VHDL semantics: The simulation cycle (3)

d) $\forall$ P sensitive to s: if event on s in current cycle: P resumes.

e) Each ... process that has resumed in the current simulation cycle is executed until it suspends*.

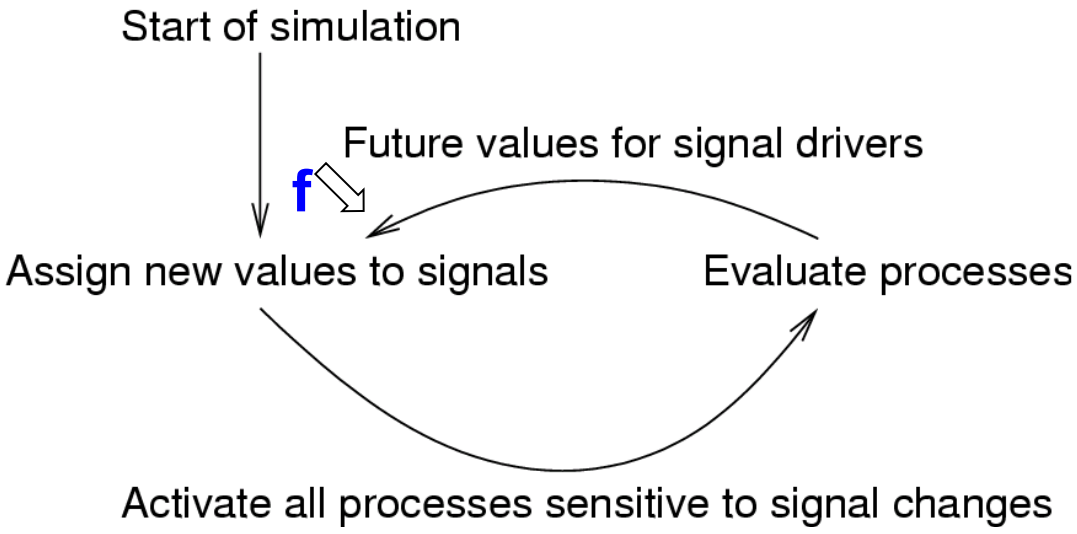
*Generates future values for signal drivers.



VHDL semantics: The simulation cycle (4)

f) Time T_n of the next simulation cycle = earliest of
- time‘high (end of simulation time).
- The next time at which a driver becomes active
- The next time at which a process resumes (determined by **wait for** statements).

Next simulation cycle (if any) will be a delta cycle if $T_n = T_c$.



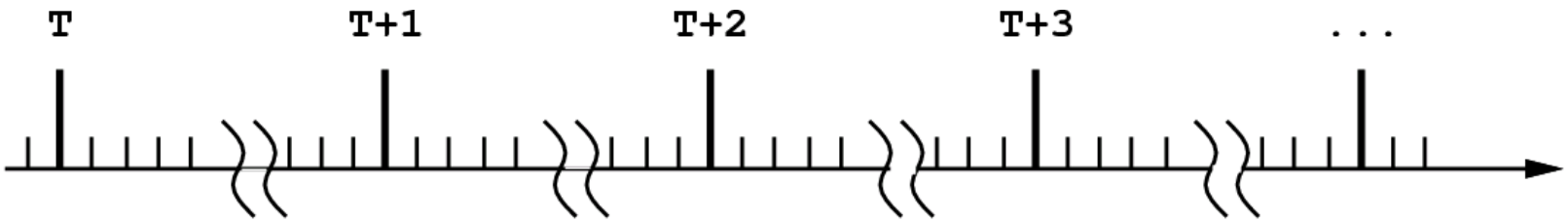
δ-simulation cycles

…

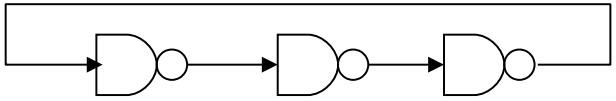
Next simulation cycle (if any) will be a delta cycle if $T_n = T_c$.

Delta cycles are generated for delay-less models.

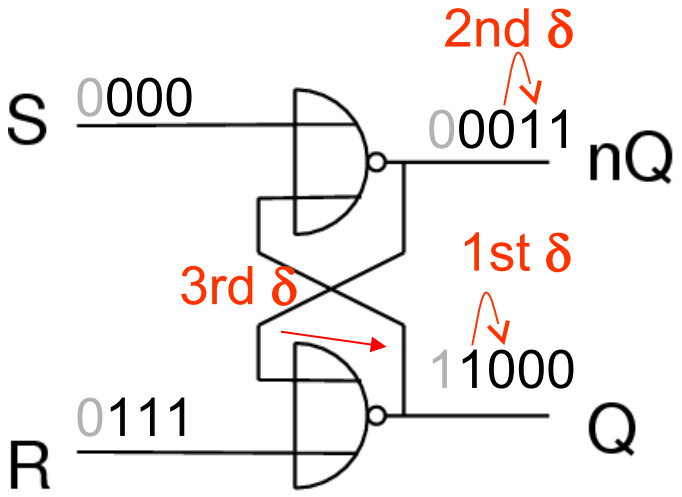
There is an arbitrary number of δ cycles between any 2 physical time instants:



In fact, simulation of delay-less hardware loops might not terminate (don't even advance T_c).



δ-simulation cycles
Simulation of an RS-Flipflop



	0ns	0ns+δ	0ns+2δ	0ns+3δ
R	1	1	1	1
S	0	0	0	0
Q	1	0	0	0
nQ	0	0	1	1

```
gate1:
  process (S,Q) is
  begin
    nQ <= S nor Q;
  end process gate1;

gate2:
  process (R,nQ) is
  begin
    Q <= R nor nQ;
  end process gate2;
```

δ cycles reflect the fact that no real gate comes with zero delay.
☞ should delay-less signal assignments be allowed at all?

δ-simulation cycles and determinate simulation semantics

Semantics of

```
a <= b;
b <= a; ?
```

Separation into 2 simulation phases results in determinate semantics (☞ StateMate).

Einschub: VHDL

– Beginn –

Gliederung

VHDL

VHSIC **H**ardware **D**escription **L**anguage
Very **H**igh **S**peed **I**ntegrated **C**ircuit

- digitale Systeme
  - Modellierung/Beschreibung
  - Simulation
  - Dokumentation
- Komponenten
  - Standard ICs
  - anwendungsspezifische Schaltungen: ASICs, FPGAs
  - Systemumgebung: Protokolle, Software ...

VHDL (cont.)

- Abstraktion
  - von der Spezifikation – Algorithmen und Protokolle
  - über die Implementation – Register-Transfer Modelle
  - bis hin zum fertigen Entwurf – Netzliste mit Backannotation

  $\Rightarrow$ VHDL durchgängig einsetzbar

  $\Rightarrow$ Simulation immer möglich

Entwicklung

- 1983 vom DoD initiiert
- 1987 IEEE Standard IEEE 1076
- 2004 IEC Standard IEC 61691-1-1
- regelmäßige Überarbeitungen: VHDL'93, VHDL'02, VHDL'08, VHDL'11

VHDL (cont.)

Erweiterungen

- Modellierung und Zellbibliotheken — IEC 61691-2, IEC 61691-5
- Hardwaresynthese — IEC 61691-3-3, IEC 62050
- mathematische Typen und Funktionen — IEC 61691-3-2
- analoge Modelle und Simulation — IEC 61691-6

Links

- `https://tams.informatik.uni-hamburg.de/research/vlsi/vhdl`
- `http://www.vhdl-online.de`
- `http://www.vhdl.renerta.com`
- `http://www.itiv.kit.edu/english/721.php`
- `http://www.asic-world.com/vhdl`
- `https://en.wikipedia.org/wiki/VHDL`
- `https://de.wikipedia.org/wiki/Very_High_Speed_Integrated_Circuit_Hardware_Description_Language`

VHDL – sequenziell

- Typen, Untertypen, Alias-Deklarationen
  - skalar `integer`, `real`, `character`, `boolean`, `bit`, Aufzählung
  - komplex `line`, `string`, `bit_vector`, Array, Record
  - Datei `text`, File
  - Zeiger Access
  - strikte Typbindung
  - Konvertierungsfunktionen
- Objekte `constant`, `variable`, `file`
- Operatoren
  1. logisch `and`, `or`, `nand`, `nor`, `xor`, `xnor`
  2. relational =, /=, <, <=, >, >=
  3. schiebend `sll`, `srl`, `sla`, `sra`, `rol`, `ror`
  4. additiv +, -, &
  5. vorzeichen +, -
  6. multiplikativ *, /, `mod`, `rem`
  7. sonstig **, `abs`, `not`

VHDL – sequenziell (cont.)

- Anweisungen
  - Zuweisung `:=`, `<=`
  - Bedingung `if`, `case`
  - Schleifen `for`, `while`, `loop`, `exit`, `next`
  - Zusicherung `assert`, `report`
  - ...
- Sequenzielle Umgebungen
  - Prozesse `process`
  - Unterprogramme `procedure`, `function`
  - lokale Gültigkeitsbereiche
  - Deklarationsteil: definiert Typen, Objekte, Unterprogramme
    Anweisungsteil: Codeanweisungen sequenziell ausführen

⇒ Imperative sequenzielle Programmiersprache (z.B. Pascal)

- Beliebige Programme *ohne Bezug zum Hardwareentwurf* möglich
- Beispiel: Datei einlesen, verlinkte Liste erzeugen ...

VHDL – sequenziell (cont.)


```
...
type      LIST_T;
type      LIST_PTR          is  access LIST_T;
type      LIST_T            is  record KEY   : integer;
                                       LINK  : LIST_PTR;
                              end record LIST_T;

constant  INPUT_ID          : string   := "inFile.dat";
file      DATA_FILE         : text;
variable  DATA_LINE         : line;
variable  LIST_P, TEMP_P    : LIST_PTR := null;

procedure READ_DATA is     -- Datei einlesen, Liste aufbauen
  variable  KEY_VAL         : integer;
  variable  FLAG            : boolean;
begin
  file_open (DATA_FILE, INPUT_ID, read_mode);
  L1: while not endfile(DATA_FILE) loop
      readline(DATA_FILE, DATA_LINE);
      L2: loop
          read(DATA_LINE, KEY_VAL, FLAG);
          if FLAG  then   TEMP_P := new LIST_T'(KEY_VAL, LIST_P);
                          LIST_P := TEMP_P;
                   else   next L1;
          end if;
      end loop L2;
  end loop L1;
  file_close(DATA_FILE);
end procedure READ_DATA;
...
```

VHDL – konkurrent

- ähnlich ADA'83
- Konkurrenter Code
  - mehrere Prozesse
  - Prozeduraufrufe
  - Signalzuweisung `<=`
    - bedingt `<= ... when ...`
    - selektiv `with ... select ... <= ...`
  - Zusicherung `assert`
  - modelliert gleichzeitige Aktivität der Hardwareelemente
- Synchronisationsmechanismus für Programmlauf / Simulation
  - Objekt signal
  - Signale verbinden konkurrent arbeitende „Teile“ miteinander
  - Entsprechung in Hardware: Leitung

VHDL – Simulation

- Semantik der Simulation im Standard definiert: Simulationszyklus
- konkurrent aktive Codefragmente
  - Prozesse + konkurrente Anweisungen + Instanzen (in Hierarchien)
  - durch Signale untereinander verbunden

„Wie werden die Codeteile durch einen sequenziellen Simulationsalgorithmus abgearbeitet?“

- Signaltreiber: Liste aus Wert-Zeit Paaren

`S: integer` ←

NOW	+5 ns	+12 ns	+15 ns	+21 ns	+27 ns	Zeitpunkt
2	7	3	12	8	-3	Wert

- Simulationsereignis
  - Werteänderung eines Signals
  - (Re-) Aktivierung eines Prozesses nach Wartezeit

VHDL – Simulation (cont.)

⇒ Ereignisgesteuerte Simulation

- theoretisches Modell
- veranschaulicht Semantik für den VHDL-Benutzer
- praktische Implementation durch Simulationsprogramme weicht (aus Performanzgründen) in der Regel davon ab
- vielfältige Optimierungsmöglichkeiten
  - compilierende Simulation
  - Ausnutzen von Datenabhängigkeiten
  - zyklenbasierte Simulation
  - Hardwareemulation
  - mixed-mode Simulation
  - ...

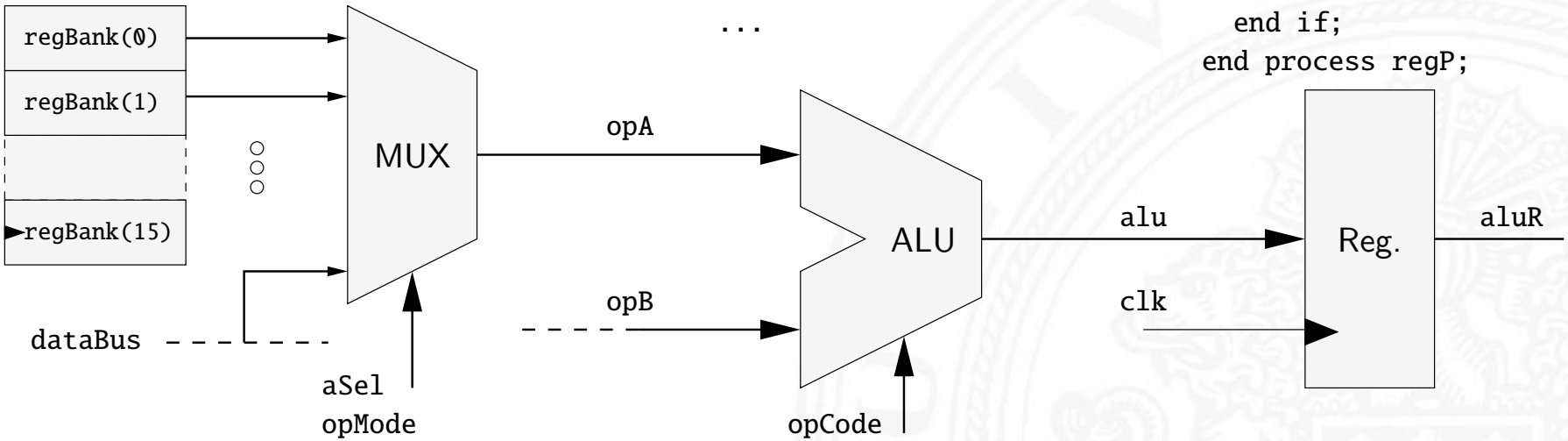
Ereignisgesteuerte Simulation

Beispiel: Datenpfad

```
opA <= regBank(aSel)
         when opMode=regM else
       dataBus;
```

```
with opCode select
alu <= opA  +  opB when opcAdd,
       opA  -  opB when opcSub,
       opA and opB when opcAnd,
       ...
```

```
regP: process (clk) is
begin
  if rising_edge(clk) then
    aluR <= alu;
  end if;
end process regP;
```



Ereignisgesteuerte Simulation – Zyklus 1

1. Simulationsereignis

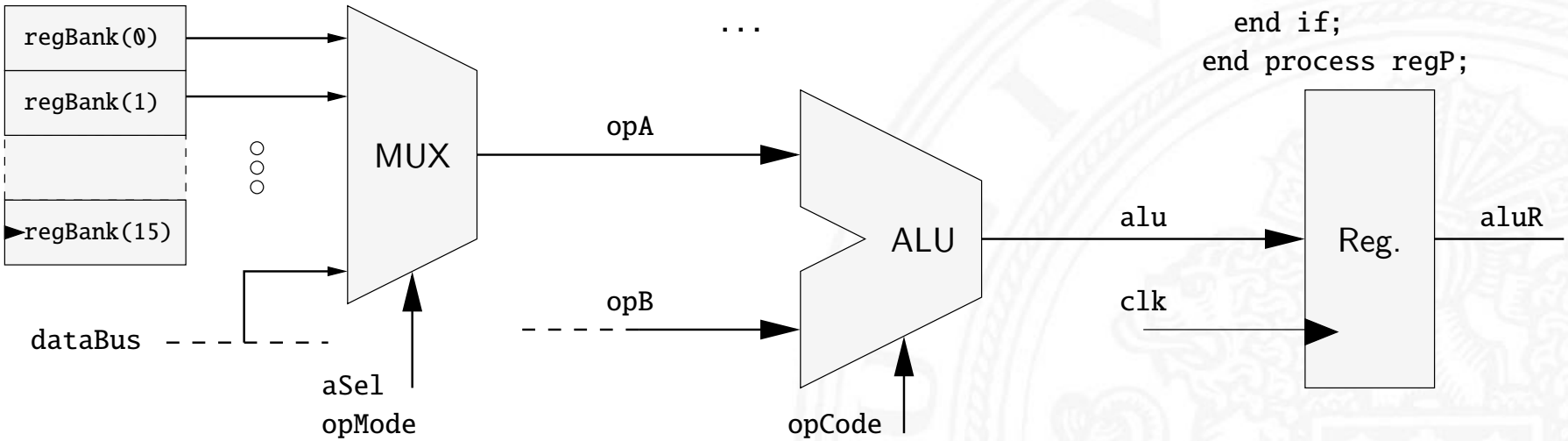
Ereignisliste

NOW	aSel	1
	opMode	regM
	opCode	opcAdd
+10 ns	clk	'1'
...		

```
opA <= regBank(aSel)
         when opMode=regM else
       dataBus;
```

```
with opCode select
alu <= opA  +  opB when opcAdd,
       opA  -  opB when opcSub,
       opA and opB when opcAnd,
       ...
```

```
regP: process (clk) is
begin
  if rising_edge(clk) then
    aluR <= alu;
  end if;
end process regP;
```



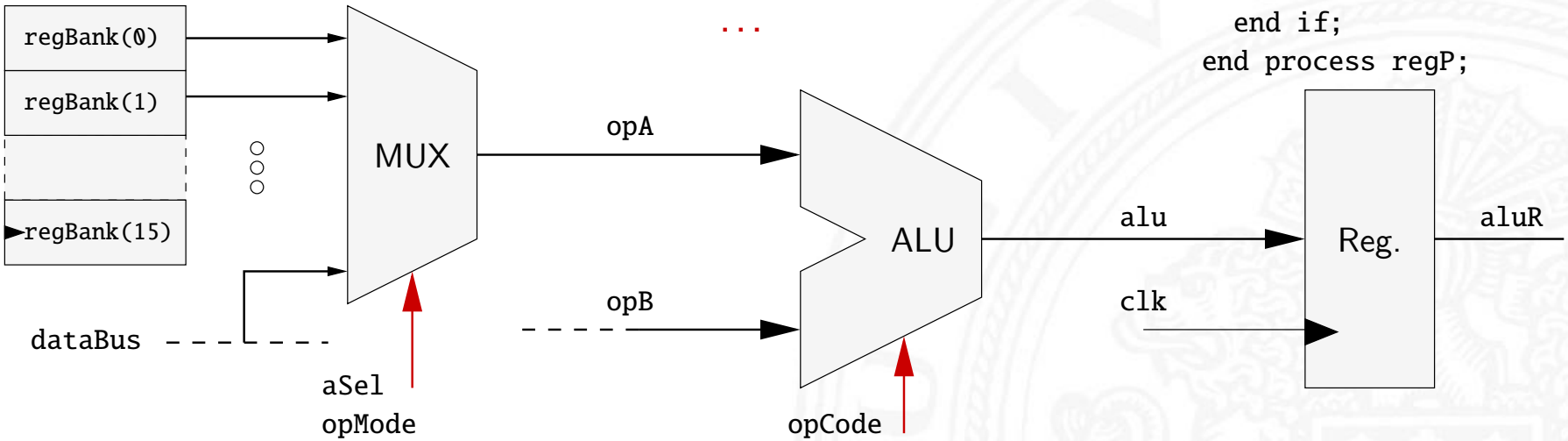
Ereignisgesteuerte Simulation – Zyklus 1

1. Simulationsereignis
2. Prozessaktivierung

```
opA <= regBank(aSel)
         when opMode=regM else
       dataBus;
```

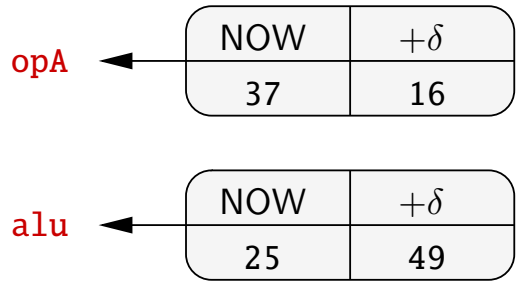
```
with opCode select
alu <= opA  +  opB when opcAdd,
       opA  -  opB when opcSub,
       opA and opB when opcAnd,
       ...
```

```
regP: process (clk) is
begin
  if rising_edge(clk) then
    aluR <= alu;
  end if;
end process regP;
```



Ereignisgesteuerte Simulation – Zyklus 1

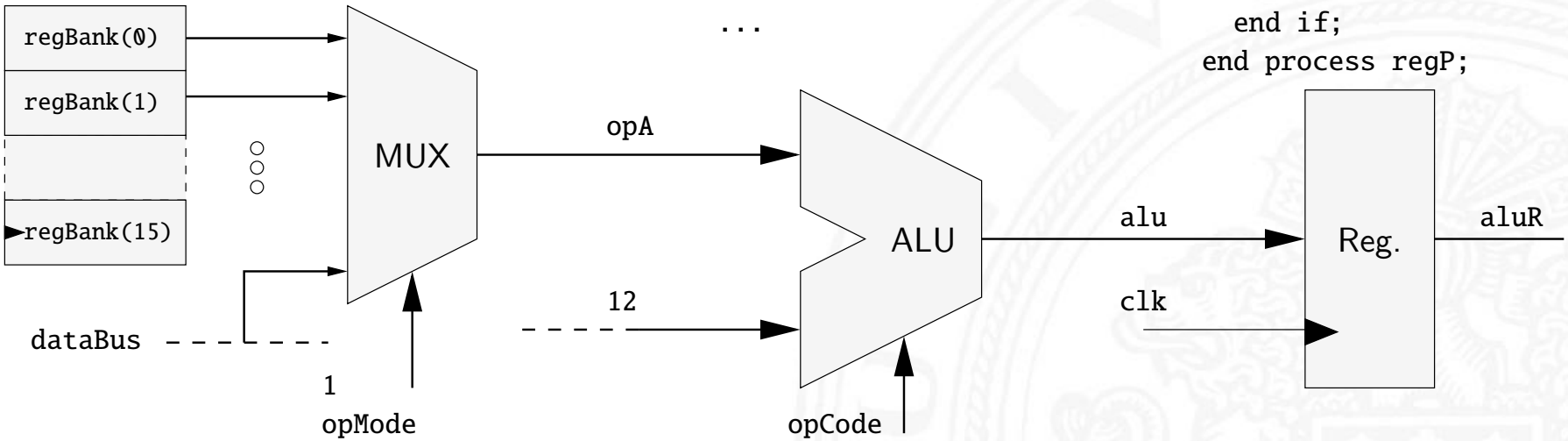
1. Simulationsereignis
2. Prozessaktivierung
3. Aktualisierung der Signaltreiber




```
opA <= regBank(aSel)
         when opMode=regM else
       dataBus;
```

```
with opCode select
alu <= opA  +  opB when opcAdd,
       opA  -  opB when opcSub,
       opA and opB when opcAnd,
       ...
```

```
regP: process (clk) is
begin
  if rising_edge(clk) then
    aluR <= alu;
  end if;
end process regP;
```



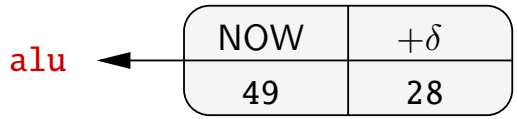
Ereignisgesteuerte Simulation – Zyklus 2

Zeitschritt: $+\delta$

Simulationsereignisse

$+\delta$	opA	16
	alu	49
$+10$ ns	clk	'1'
...		

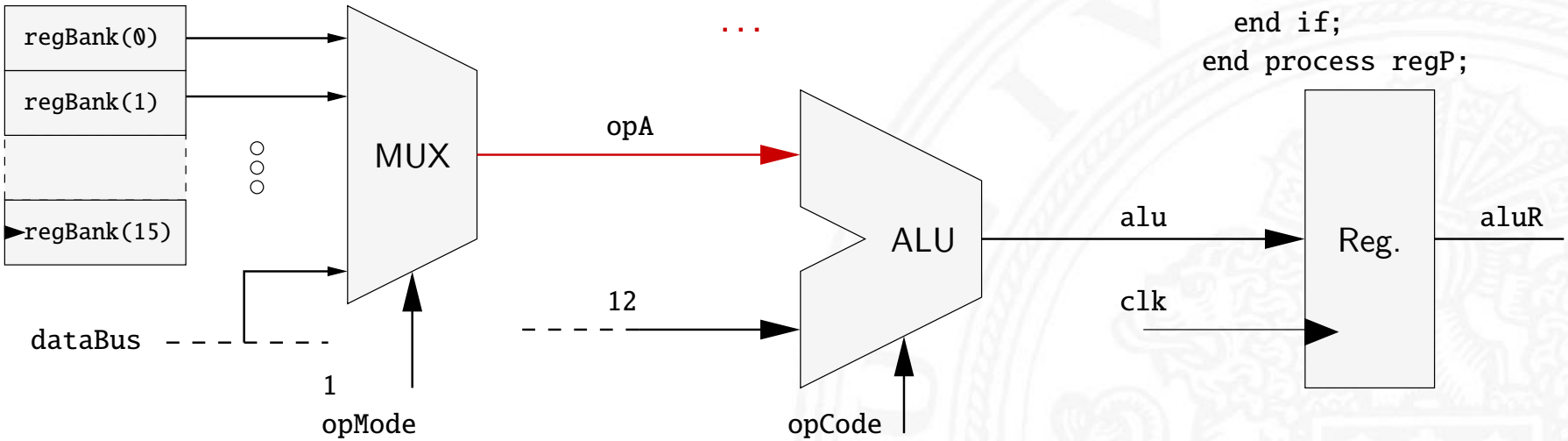
Signaltreiber




```
opA <= regBank(aSel)
         when opMode=regM else
       dataBus;
```

```
with opCode select
alu <= opA  +  opB when opcAdd,
       opA  -  opB when opcSub,
       opA and opB when opcAnd,
       ...
```

```
regP: process (clk) is
begin
  if rising_edge(clk) then
    aluR <= alu;
  end if;
end process regP;
```



Ereignisgesteuerte Simulation – Zyklus 3

Zeitschritt: +10 ns

Simulationsereignisse

+10 ns clk '1'

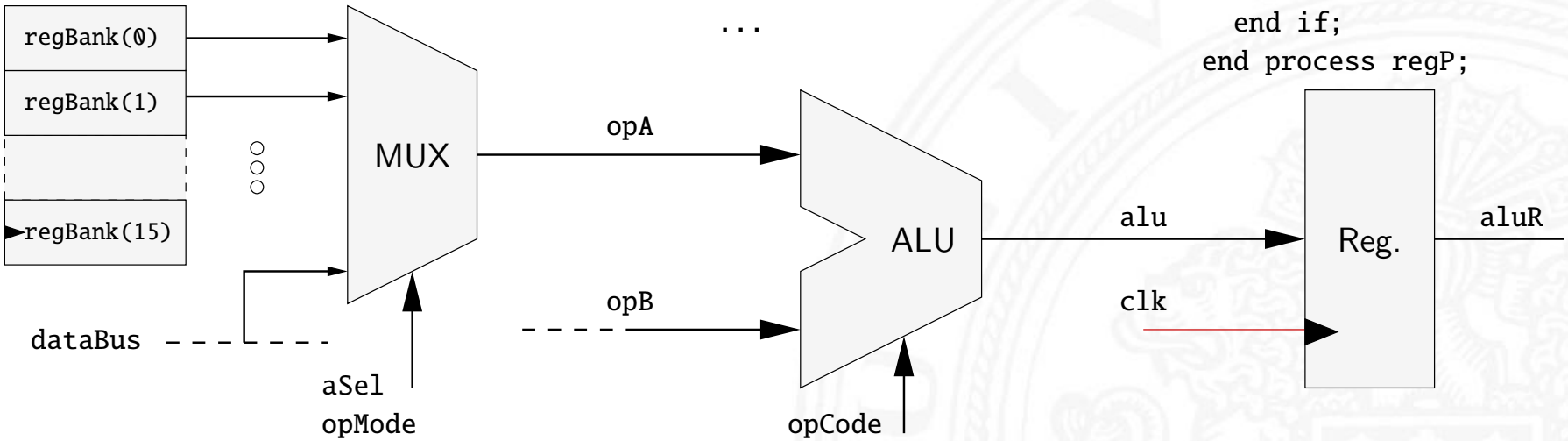
...

Signaltreiber


```
opA <= regBank(aSel)
         when opMode=regM else
       dataBus;
```

```
with opCode select
alu <= opA  +  opB when opcAdd,
       opA  -  opB when opcSub,
       opA and opB when opcAnd,
       ...
```

```
regP: process (clk) is
begin
  if rising_edge(clk) then
    aluR <= alu;
  end if;
end process regP;
```



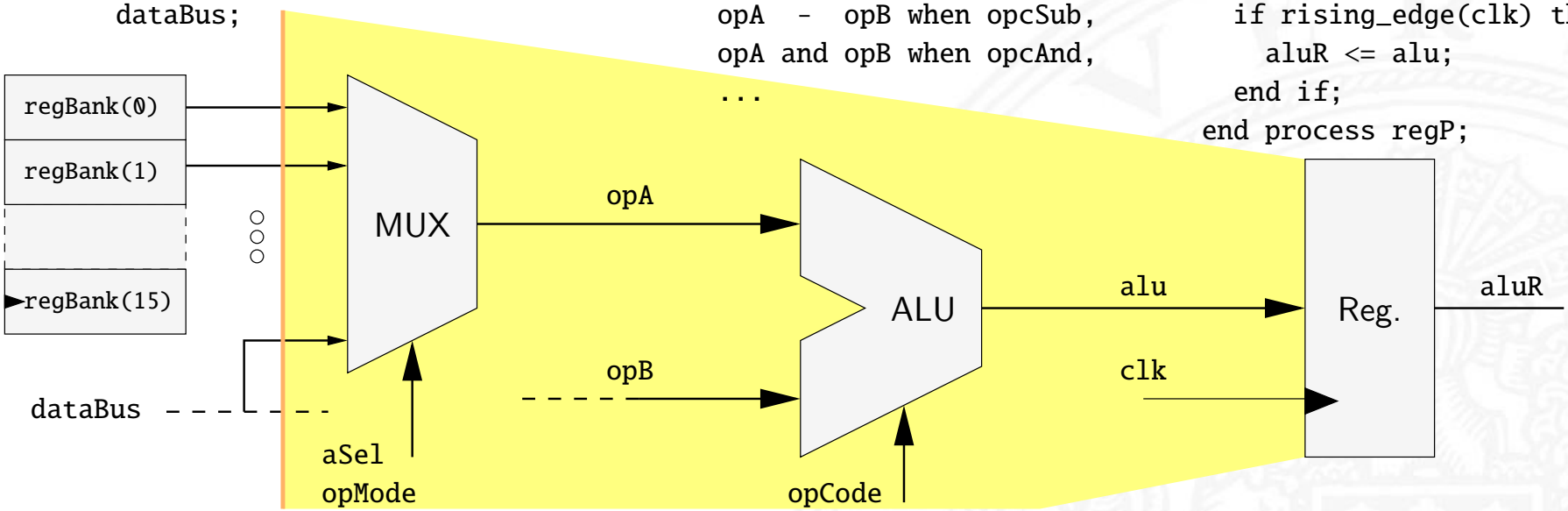
Zyklenbasierte Simulation

- Diskretes Zeitraster: Takt bei Register-Transfer Code
- In jedem Zyklus werden alle Beschreibungen simuliert
- Sequenzialisierung der Berechnung entsprechend den Datenabhängigkeiten

```
opA <= regBank(aSel)
         when opMode=regM else
       dataBus;
```

```
with opCode select
alu <= opA  +  opB when opcAdd,
       opA  -  opB when opcSub,
       opA and opB when opcAnd,
       ...
```

```
regP: process (clk) is
begin
  if rising_edge(clk) then
    aluR <= alu;
  end if;
end process regP;
```



VHDL – Simulationszyklus

Semantik der Simulation

- Kausalität
  1. Simulationsereignis — Simulationszyklus$_n$
  2. Aktivierung des konkurrenten Codes
  3. Signalzuweisungen in der Abarbeitung

  .....

  4. Erneute Signaländerung — Simulationszyklus$_{n+1}$
- Trennung der Zyklen

  $\Rightarrow$ Simulation ist *unabhängig von der sequenziellen Abarbeitungsreihenfolge* durch den Simulator

  $\Rightarrow$ auch bei *mehreren Events* in einem Zyklus, bzw. bei *mehrfachen Codeaktivierungen* pro Event

VHDL – Simulationszyklus (cont.)

Prozesse / Umgebungen von sequenziellem Code

- ständig aktiv $\Rightarrow$ Endlosschleife

1. Sensitiv zu Signalen
   - Aktivierung, bei Ereignis eines Signals
   - Abarbeitung aller Anweisungen bis zum Prozessende

```
ALU_P: process (A, B, ADD_SUB) is
begin
  if ADD_SUB  then  X <= A + B;
              else  X <= A - B;
  end if;
end process ALU_P;
```

2. explizite `wait`-Anweisungen
   - Warten bis Bedingung erfüllt ist
   - Abarbeitung aller Anweisungen bis zum nächsten `wait`
   - Prozessende wird „umlaufen“ (Ende einer Schleife)

VHDL – Simulationszyklus (cont.)

Beispiel: Erzeuger / Verbraucher

```
...
signal C_READY, P_READY : boolean        -- Semaphore
signal CHANNEL          : ...            -- Kanal

PRODUCER_P: process is                   -- Erzeuger
begin
  P_READY   <= false;
  wait until C_READY;
  CHANNEL   <= ...                       -- generiert Werte
  P_READY   <= true;
  wait until not C_READY;
end process PRODUCER_P;

CONSUMER_P: process is                   -- Verbraucher
begin
  C_READY   <= true;
  wait until P_READY;
  C_READY   <= false;
        ... <= CHANNEL;                  -- verarbeitet Werte
  wait until not P_READY;
end process CONSUMER_P;
```

VHDL – Simulationszyklus (cont.)

Signalzuweisungen im sequenziellen Kontext

- sequenzieller Code wird nach der Aktivierung bis zum Prozessende / zum `wait` abgearbeitet
- Signalzuweisungen werden erst in folgenden Simulationszyklen wirksam, frühestens im nächsten δ-Zyklus

$\Rightarrow$ eigene Zuweisungen sind im sequenziellen Kontext des Prozesses nicht sichtbar

```
process ...
  ...
  if SWAP = '1' then   -- Werte tauschen
    B <= A;            -- B = 'altes' A
    A <= B;            -- A = 'altes' B
  end if;
```

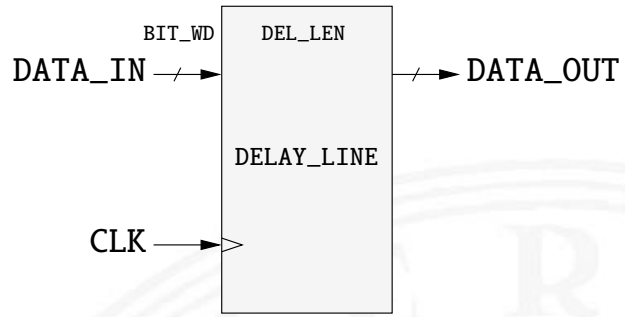
```
process ...
  ...
  NUM <= 5;            -- Zuweisung
  ...
  if NUM > 0 then      -- ggf. /= 5 !!!
    ...
```

Entwurfsspezifische Eigenschaften

- Strukturbeschreibungen / Hierarchie
  - Instanzen `component` `configuration`
  - Schnittstellen `entity`
  - Versionen und Alternativen (*exploring the design-space*)
    `architecture` `configuration`
- Management von Entwürfen
  - Bibliotheken `library`
  - Code-Reuse `package`
  - VHDL-Erweiterungen: Datentypen, Funktionen ...
  - Gatterbibliotheken
  - spezifisch für EDA-Tools (**E**lectronic **D**esign **A**utomation)
  - eigene Erweiterungen, firmeninterne Standards ...

VHDL – Entity

- Beschreibung der Schnittstelle „black-box“
  - mit Parametern `generic`
  - mit Ein- / Ausgängen `port`



parametrierbare Verzögerungsleitung

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity DELAY_LINE is
generic(BIT_WD          : integer range 2 to 64 := 16;
        DEL_LEN         : integer range 2 to 16 := 16);
port (  CLK             : in  std_logic;
        DATA_IN         : in  signed(BIT_WD-1 downto 0);
        DATA_OUT        : out signed(BIT_WD-1 downto 0));
end entity DELAY_LINE;
```

VHDL – Architecture

- Implementation einer Entity
- mehrere Architekturen möglich $\Rightarrow$ Alternativen
- Deklarationsteil: Typen, Signale, Unterprogramme, Komponenten, Konfigurationen
  Anweisungsteil: Prozesse, konkurrente Anweisungen, Instanzen

```
architecture BEHAVIOR of DELAY_LINE is
  type    DEL_ARRAY_TY is array (1 to DEL_LEN) of signed(BIT_WD-1 downto 0);
  signal  DEL_ARRAY     : DEL_ARRAY_TY;
begin
  DATA_OUT      <= DEL_ARRAY(DEL_LEN);
  REG_P: process (CLK) is
  begin
    if rising_edge(CLK) then
      DEL_ARRAY <= DATA_IN & DEL_ARRAY(1 to DEL_LEN-1);
    end if;
  end process REG_P;
end architecture BEHAVIOR;
```

VHDL – strukturell

- Hierarchie
  - funktionale Gliederung des Entwurfs
  - repräsentiert Abstraktion
- Instanziierung von Komponenten
  1. Komponentendeklaration `component`
     - im lokalen Kontext
     - in externen Packages
  2. Instanz im Code verwenden
     - im Anweisungsteil der Architecture
     - Mapping von Ein- und Ausgängen / Generic-Parametern
  3. Bindung: Komponente $\Leftrightarrow$ Entity+Architecture
     - lokal im Deklarationsteil `for ... use ...`
     - als eigene VHDL-Einheit `configuration`

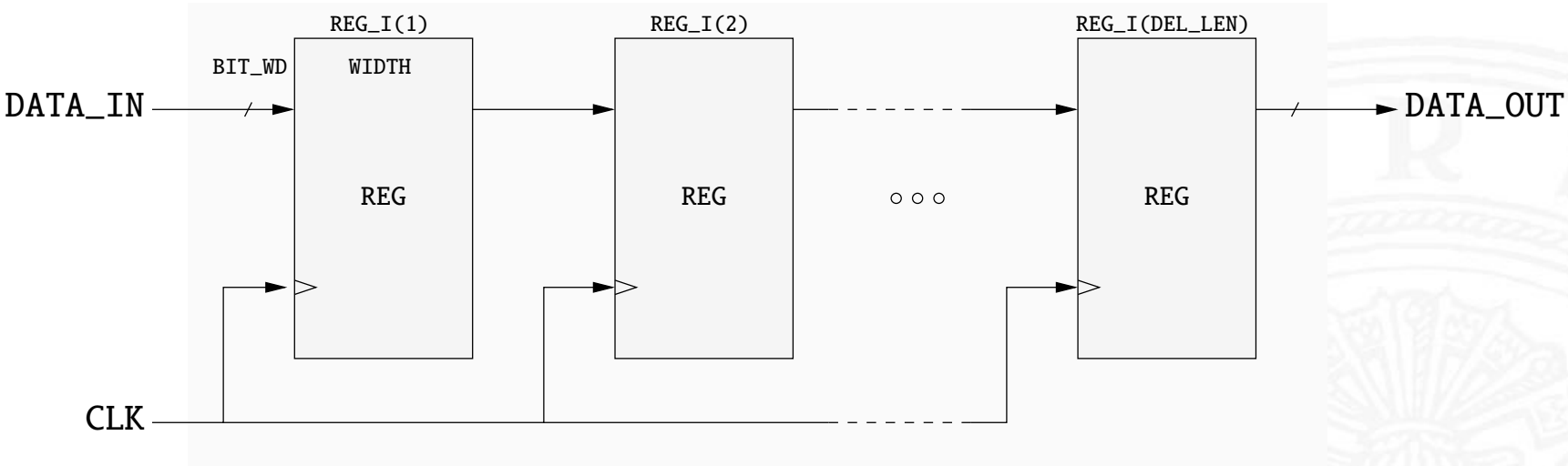
VHDL – strukturell (cont.)

- Komponente: lokale Zwischenstufe im Bindungsprozess
  - andere Bezeichner, Schnittstellen (Ports und Generics)
  - bei Bibliothekselementen wichtig
  - 2-stufige Abbildung
    1. Instanz in Architektur $\Leftrightarrow$ Komponente
    2. Komponente $\Leftrightarrow$ Entity+Architecture
  - „Default“-Konfiguration
    - gleiche Bezeichner und Deklaration
    - zuletzt (zeitlich) analysierte Architektur
- strukturierende Anweisungen
  - Gruppierung `block`
  - konkurrenten Code (Prozesse, Anweisungen, Instanzen . . .)
    - bedingt ausführen `if ... generate ...`
    - wiederholt ausführen `for ... generate ...`

VHDL – strukturell (cont.)

Beispiel als Strukturbeschreibung von Registern: `REG`

- Die Register sind als eigene VHDL-Entities / -Architekturen woanders definiert



VHDL – strukturell (cont.)


```
architecture STRUCTURE of DELAY_LINE is
  component REG is                                  -- 1. Komponentendeklaration
  generic( WIDTH        : integer range 2 to 64);
  port   ( CLK          : in std_logic;
           D_IN         : in  signed(WIDTH-1 downto 0);
           D_OUT        : out signed(WIDTH-1 downto 0));
  end component REG;
  type    DEL_REG_TY is array (0 to DEL_LEN) of signed(BIT_WD-1 downto 0);
  signal  DEL_REG       : DEL_REG_TY;
begin
  DEL_REG(0)    <= DATA_IN;
  DATA_OUT      <= DEL_REG(DEL_LEN);
  GEN_I: for I in 1 to DEL_LEN generate
    REG_I: REG generic map (WIDTH => BIT_WD)    -- 2. Instanziierung
                port map (CLK, DEL_REG(I-1), DEL_REG(I));
  end generate GEN_I;
end architecture STRUCTURE;
```

VHDL – strukturell (cont.)

Konfigurationen

1. Auswahl der Architektur durch eindeutigen Bezeichner

```
configuration DL_BEHAVIOR of DELAY_LINE is
for BEHAVIOR
end for;
end configuration DL_BEHAVIOR;
```

2. Bindung von Instanzen in der Hierarchie

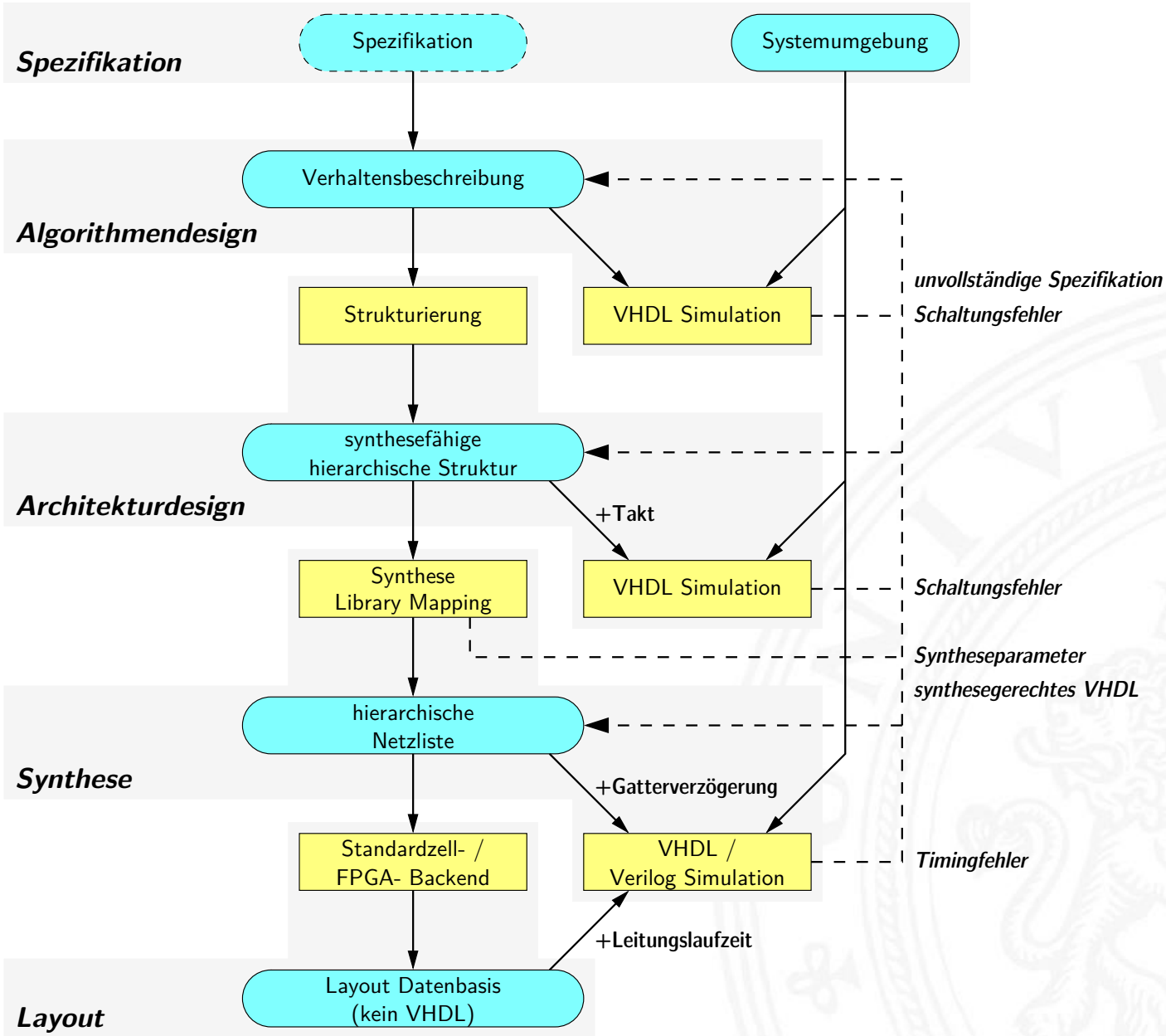
```
configuration DL_STRUCTURE of DELAY_LINE is
for STRUCTURE
  for GEN_I
    for all: REG use entity work.REG(BEHAVIOR);
    end for;
  end for;
end for;
end configuration DL_STRUCTURE;
```

VHDL – Entwurfsmethodik

- VHDL Abstraktion: von Algorithmen- bis zur Gatterebene
  $\Rightarrow$ eine Sprache als Ein- und Ausgabe der Synthese
- Synthese: ab der RT-Ebene
  - Abbildung von Register-Transfer Beschreibungen auf Gatternetzlisten
  - erzeugt neue Architektur, Entity bleibt

  High-Level Synthese
  - eingeschränkter „Suchraum“, feste Zielarchitektur
  - spezielle Anwendungsfälle
- Simulation
  - System-/Testumgebung als VHDL-Verhaltensmodell
  - Simulation der Netzliste durch Austausch der Architektur

VHDL-basierter Entwurfsablauf



VHDL – Entwurfsmethodik (cont.)

- Modellierung der Systemumgebung
- Simulation auf allen Abstraktionsebenen

Algorithmenebene	Auswahl verschiedener Algorithmen keine Zeitmodelle
RT-Ebene	Datenabhängigkeiten: Ein-/Ausgabe (Protokolle) Taktbasierte Simulation
Gatterebene	+ Gatterverzögerungen
Layout	+ Leitungslaufzeiten

- Synthese ab der Register-Transfer Ebene

– Ende –

Summary

Discrete event models

- Queue of future events, fetch and execute cycle, commonly used in HDLs
- processes model HW concurrency
- signals model communication
- **wait**, sensitivity lists
- the VHDL simulation cycle
  - $\forall$ δ cycles, determinate simulation

Multiple-valued logic

- General CSA approach
- Application to IEEE 1164

VHDL: Evaluation

- Behavioral hierarchy (procedures and functions),
- Structural hierarchy: through structural architectures, but no nested processes,
- No specification of non-functional properties,
- No object-orientation,
- Static number of processes,
- Complicated simulation semantics,
- Too low level for initial specification,
- Good as an intermediate “Esperanto“ or ”assembly” language for hardware generation.

Using C for ES Design: Motivation

- Many standards (e.g. the GSM and MPEG-standards) are published as C programs
  - ☞ Standards have to be translated if special hardware description languages have to be used
- The functionality of many systems is provided by a mix of hardware and software components
  - ☞ Simulations require an interface between hardware and software simulators unless the same language is used for the description of hardware and software

☞ Attempts to describe software and hardware in the same language. Easier said than implemented.
Various C dialects used for hardware description.

Drawbacks of a C/C++ Design Flow

- C/C++ is *not* created to design hardware !
- C/C++ does not support
  - Hardware style communication - Signals, protocols
  - Notion of time - Clocks, time sequenced operations
  - Concurrency - Hardware is concurrent, operates in ||
  - Reactivity - Hardware is reactive, responds to stimuli, interacts with its environment (requires handling of exceptions)
  - Hardware data types - Bit type, bit-vector type, multi-valued logic types, signed and unsigned integer types, fixed-point types
- Missing links to hardware during debugging

SystemC: Required features

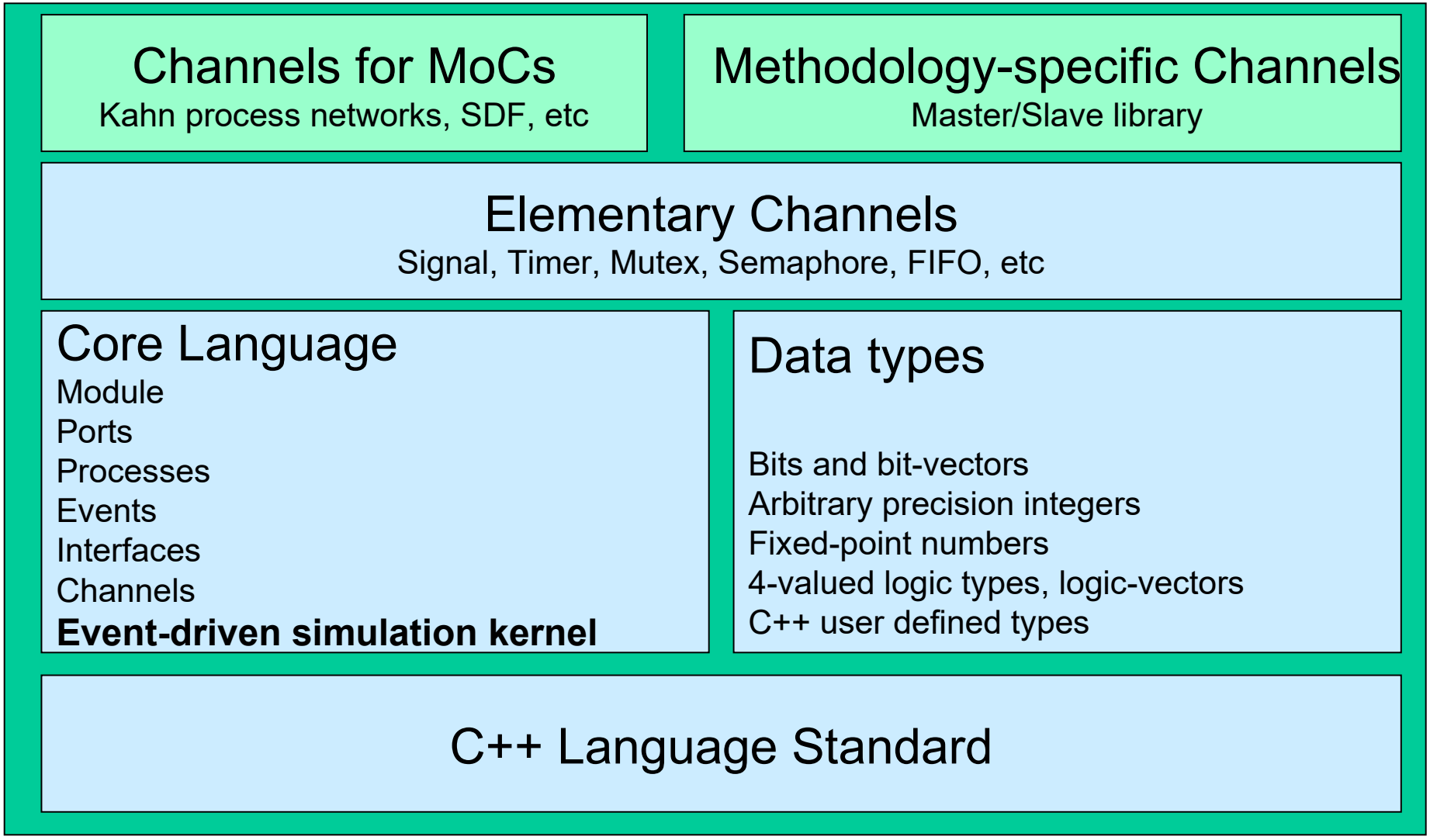
Requirements, solutions for modeling HW in a SW language:

- **C++ class library including required functions.**
- **Concurrency:** via processes, controlled by sensitivity lists* and calls to wait primitives.
- **Time:** Floating point numbers in SystemC 1.0.
  Integer values in SystemC 2.0;
  Includes units such as ps, ns, µs etc*.
- **Support of bit-datatypes**: bitvectors of different lengths; 2- and 4-valued logic; built-in resolution*)
- **Communication:** plug-and-play (pnp) channel model, allowing easy replacement of intellectual property (IP)
- Determinate behavior not guaranteed.

* Good to know VHDL ☺

SystemC language architecture

https://www.accellera.org/community/systemc



Transaction-based modeling

Definition: "***Transaction-level modeling*** *(TLM) is a high-level approach to modeling digital systems where details of communication among modules are separated from the details of the implementation of functional units or of the communication architecture.*
Communication mechanisms such as buses or FIFOs are modeled as channels, and are presented to modules using SystemC interface classes. Transaction requests take place by calling interface functions of these channel models, which encapsulate low-level details of the information exchange.
At the transaction level, the emphasis is more on the functionality of the data transfers - what data are transferred to and from what locations - and less on their actual implementation, that is, on the actual protocol used for data transfer.
This approach makes it easier for the system-level designer to experiment, for example, with different bus architectures (all supporting a common abstract interface) without having to recode models that interact with any of the buses, provided these models interact with the bus though the common interface."

Grötker et al., 2002

Verilog

HW description language competing with VHDL

Standardized:

- IEEE 1364-1995 (Verilog version 1.0)
- IEEE 1364-2001 (Verilog version 2.0)
- Features similar to VHDL:
  - Designs described as connected entities
  - Bitvectors and time units are supported
- Features that are different:
  - Built-in support for 4-value logic and for logic with 8 strength levels encoded in two bytes per signal.
  - More features for transistor-level descriptions
  - Less flexible than VHDL.
  - More popular in the US (VHDL common in Europe)

SystemVerilog

Corresponds to Verilog versions 3.0 and 3.1. Includes:

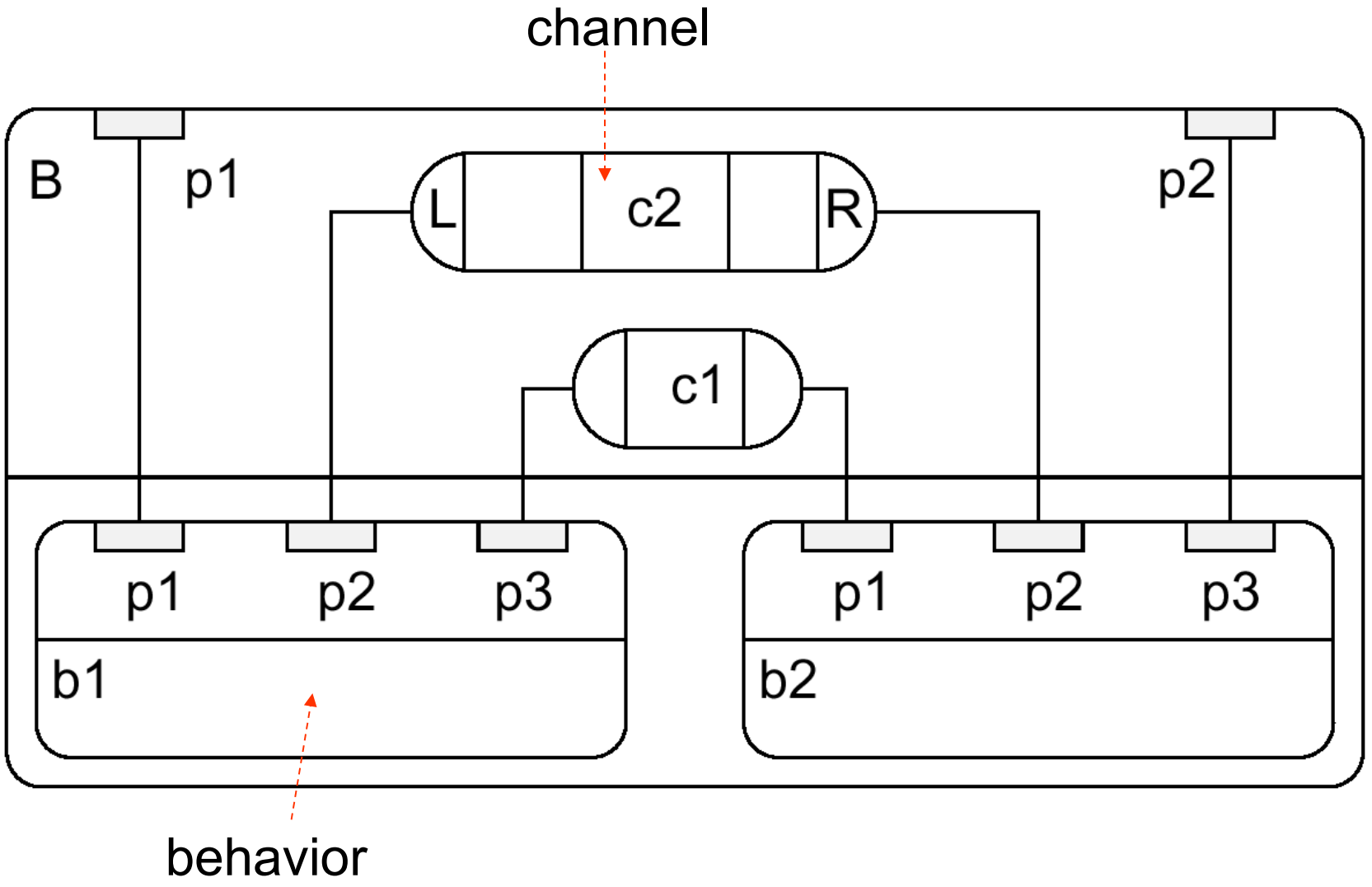
- Additional language elements for modeling behavior
- C data types such as `int`
- Type definition facilities
- Definition of interfaces of HW components as entities
- Mechanism for calling C/C++-functions from Verilog
- Limited mechanism for calling Verilog functions from C.
- Enhanced features for describing the testbench
- Dynamic process creation.
- Interprocess communication and synchronization
- Automatic memory allocation and deallocation.
- Interface for formal verification.

SpecC

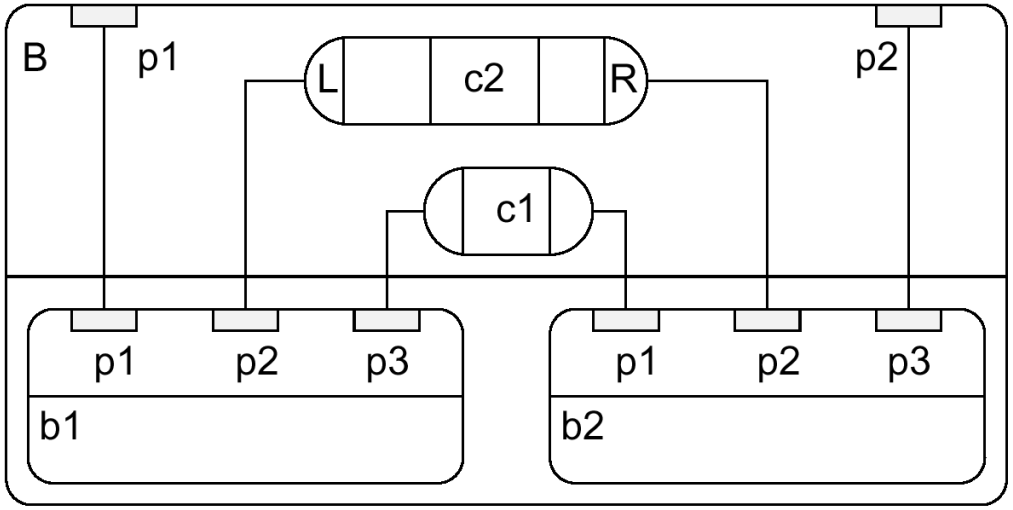
[Gajski, Dömer et. al. 2000]

- SpecC is based on the clear separation between communication and computation. Enables „*plug-and-play*“ for system components; models systems as hierarchical networks of behaviors communicating through channels.
- Consists of behaviors, channels and interfaces.
- **Behaviors** include ports, locally instantiated components, private variables and functions and a public main function.
- **Channels** encapsulate communication. Include variables and functions, used for the definition of a communication protocol.
- **Interfaces:** linking behaviors and channels.
  Declare communication protocols (*defined* in a channel).

Example

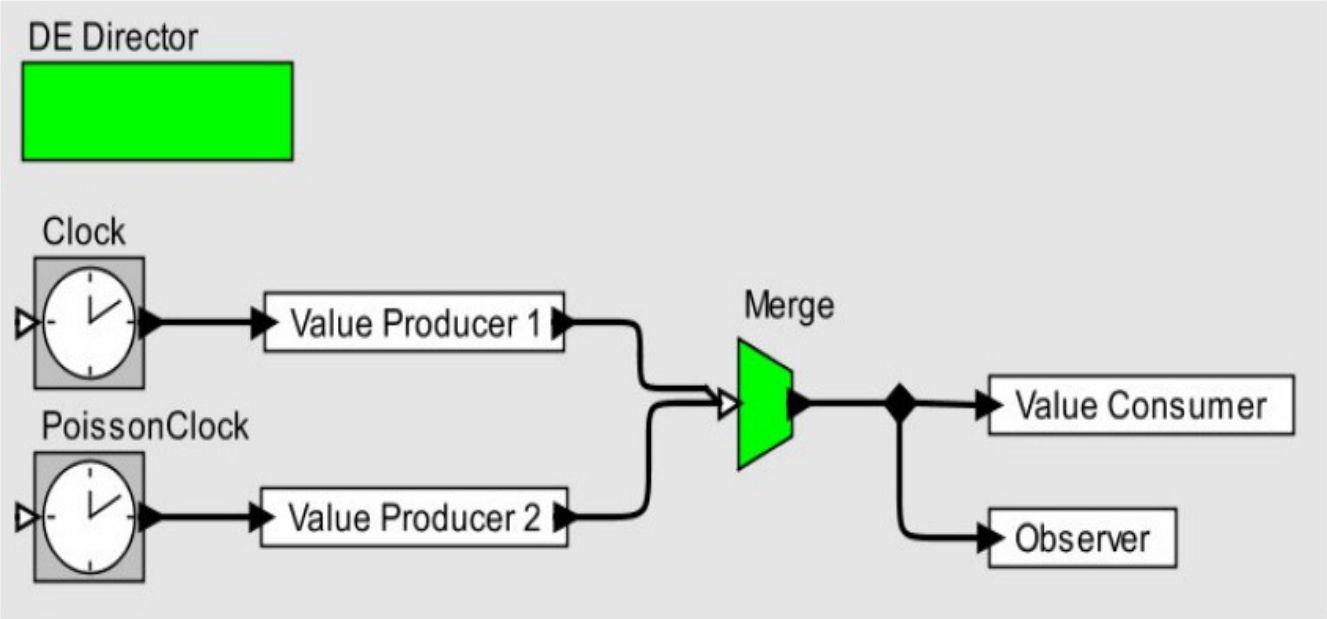


SpecC-Example




```
interface L {void Write(int x);};
interface R {int Read (void);};
channel C implements L,R
  { int Data; bool Valid;
    void Write(int x)  {Data=x; Valid=true;}
    int Read(void) {while (!Valid) waitfor(10); return (Data);}
  };
behavior B1 (in int p1, L p2, in int p3)
  {void main(void)    {/*...*/ p2.Write(p1);}  };
behavior B2 (out int p1, R p2, out int p3)
  {void main(void)   {/*...*/ p3=p2.Read();   }  };
behavior B(in int p1, out int p2)
  { int c1; C c2; B1 b1(p1,c2,c1); B2 b2 (c1,c2,p2);
    void main (void)
    {par {b1.main();b2.main();}}
  };
```

Observer Pattern using Discrete Events



Messages have a (semantic) time, and actors react to messages chronologically. Merge now becomes deterministic.

Imperative model of computation

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 11 月 07 日

Models of computation considered in this course

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorith. ☞ Computer arch.		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC* …	Only experimental systems, e.g. distributed DE in Ptolemy	
Imperative Model Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

Imperative (von-Neumann) model

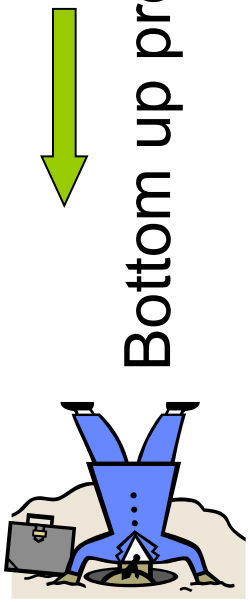
The von-Neumann model reflects the principles of operation of standard computers:

- Sequential execution of instructions (total order of instructions)

- Possible branches
- Visibility of memory locations and addresses

Example languages

- Machine languages (binary)
- Assembly languages (mnemonics)
- Imperative languages providing limited abstraction of machine languages (C, C++, Java, ….)



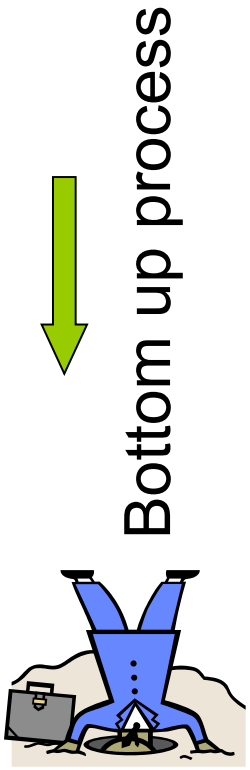
Threads/processes

Threads/processes

- Initially available only as entities managed by OS
- In most cases:
  - Context switching between threads/processes, frequently based on pre-emption
    (cooperative multi-tasking or time-triggered system rare)
- Made available to programmer as well

☞ Partitioning of applications into threads (same address space)

- Languages initially not designed for communication, but synchronization and communication is needed!
  - ☞ Access to shared memory



Communication via shared memory

Several threads access the same memory

- Very fast communication technique (no extra copying)
- Potential race conditions:

```
thread a {
  u = 1;
  if u<5 {u = u + 1; ..}
}
```

```
thread b {
  ..
 u = 5
}
```

Context switch after the test could result in u == 6.

- ☞ inconsistent results possible
- ☞ Critical sections = sections at which exclusive access to resource *r* (e.g. shared memory) must be guaranteed

Shared memory

```
thread a {
  u = 1; ..
  P(S)  //obtain mutex
  if u<5 {u = u + 1; ..}
  // critical section
  V(S)  //release mutex
}
```

```
thread b {
  ..
  P(S)  //obtain mutex
  u = 5
  // critical section
  V(S)  //release mutex
}
```

S: semaphore

P(S) grants up to n concurrent accesses to resource

n=1 in this case (mutex/lock)

V(S) increases number of allowed accesses to resource

Imperative model should be supported by:

- mutual exclusion for critical sections
- cache coherency protocols

Deadlocks

Deadlocks can happen, if the following 4 conditions are met [Coffman, 1971]:

- **Mutual exclusion**: a resource that cannot be used by >1 thread at a time
- **Hold and wait**: thread already holding resources may request new resources
- **No preemption**: Resource cannot be forcibly removed from threads, they can be released only by the holding threads
- **Circular wait**: ≥ 2 threads form a circular chain where each thread waits for a resource that the next thread in the chain holds

Techniques for turning one of these conditions false degrade performance/ increase resource requirements seriously

In non-safety-critical software, it is “ok” to ensure that deadlocks are “sufficiently” infrequent.

Problems with imperative languages and shared memory

- Specification of total order is an over-specification.
  A partial order would be sufficient.
  The total order reduces the potential for optimizations
- Preemptions at any time complicate timing analysis
- Access to shared memory leads to anomalies, that have to be pruned away by mutexes, semaphores, monitors
- Potential deadlocks
- Access to shared, protected resources leads to priority inversion (☞ chapter 4)
- Termination in general undecidable
- Timing cannot be specified and not guaranteed

Synchronous message passing: CSP

- CSP (communicating sequential processes) [Hoare, 1985]

  Rendez-vous-based communication:

  Example:

```
process A                    process B
..                           ..
var a ...                    var b ...
  a:=3;                        ...
  c!a; -- output               c?b; -- input
end                          end
```

No race conditions (!)

Synchronous message passing: Ada-rendez-vous

```
task screen_out is
  entry call_ch(val:character; x, y: integer);
  entry call_int(z, x, y: integer);
end screen_out;
task body  screen_out is
...
select
  accept call_ch ...  do ..
  end call_ch;
or
  accept call_int ... do ..
  end call_int;
end select;
```

Sending a message:

```
begin
 screen_out.call_ch('Z',10,20);
 exception
  when tasking_error =>
            (exception handling)
end;
```

Synchronous message passing: Ada

After Ada Lovelace (said to be the 1st female programmer).

US Department of Defense (DoD) wanted to avoid multitude of programming languages

- ☞ Definition of requirements
- ☞ Selection of a language from a set of competing designs (selected design based on PASCAL)

Ada'95 is object-oriented extension of original Ada.

Salient: task concept

Synchronous message passing: Using of tasks in Ada

```
procedure example1 is
   task a;
   task b;
   task body a is
      -- local declarations for a
   begin
      -- statements for a
   end a;
   task body b is
      -- local declarations for b
   begin
      -- statements for b
   end b;
begin
   -- tasks a and b will start before the
   -- first statement of the body of
   -- example 1
end;
```

Communication/synchronization

- Communication libraries can add blocking or non-blocking communication to von-Neumann languages like C, C++, Java, …
- Examples will be presented in chapter 4

Other imperative embedded languages

- **Pearl:** Designed in Germany for process control applications. Dating back to the 70s.
  Used to be popular in Europe.
  Pearl News still exists
  (in German, see http://www.real-time.de/)

- **Chill**: Designed for telephone exchange stations.
  Based on PASCAL.
  http://psc.informatik.uni-jena.de/languages/chill/chill.htm

Java

Potential benefits:

- Clean and safe language
- Supports multi-threading (no OS required?)
- Platform independence (relevant for telecommunications)

Problems:

- Size of Java run-time libraries? Memory requirements.
- Access to special hardware features
- Garbage collection time
- Non-deterministic dispatcher
- Performance problems
- Checking of real-time constraints

Overview over Java 2 Editions

"J2ME … addresses the large, rapidly growing consumer space, which covers a range of devices from tiny commodities, such as pagers, all the way up to the TV set-top box.."

Based on java.sun.com/products/cldc/wp/KVMwp.pdf

Lee's conclusion

Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans.

…

Succinct Problem Statement

Threads are wildly nondeterministic.

The programmer's job is to prune away the nondeterminism by imposing constraints on execution order (e.g., mutexes).

Improve threads?

Or replace them?

[Edward Lee (UC Berkeley), Artemis Conference, Graz, 2007]

Comparison of models

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 11 月 07 日

Models of computation considered in this course

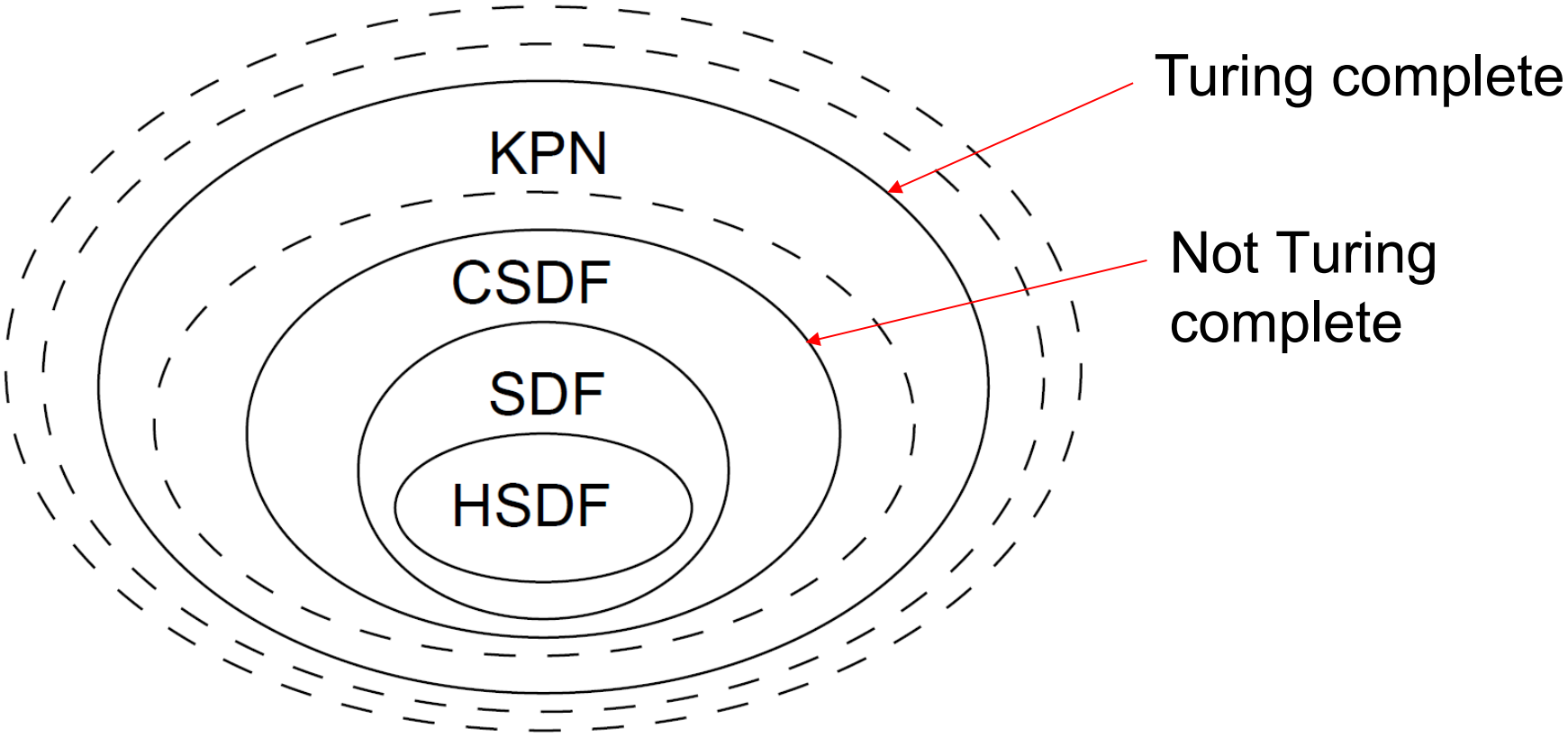
Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorith. ☞ Computer arch.		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC* …	Only experimental systems, e.g. distributed DE in Ptolemy	
Imperative Model Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

Classification by Stuijk

- **Expressiveness** and **succinctness** indicate, which systems can be modeled and how compact the are.
- **Analyzability** relates to the availability of scheduling algorithms and the need for run-time support.
- **Implementation** efficiency is influenced by the required scheduling policy and the code size.

Expressiveness of data flow models



S. Stuijk, 2007

Properties of processes/threads (1)

- **Number of processes/threads**
  static;
  dynamic (dynamically changed hardware architecture?)

- **Nesting:**
  - Nested declaration of processes

```
process {
  process {
    process {
}}}
```

  - or all declared at the same level

```
process { … }
process { … }
process { … }
```

Properties of processes/threads (2)

- Different techniques for **process creation**
  - **Elaboration in the source (c.f. ADA)**

```
declare
   process P1 …
```

  - **explicit fork and join (c.f. Unix)**

```
id = fork();
```

  - **process creation calls**

```
id = create_process(P1);
```

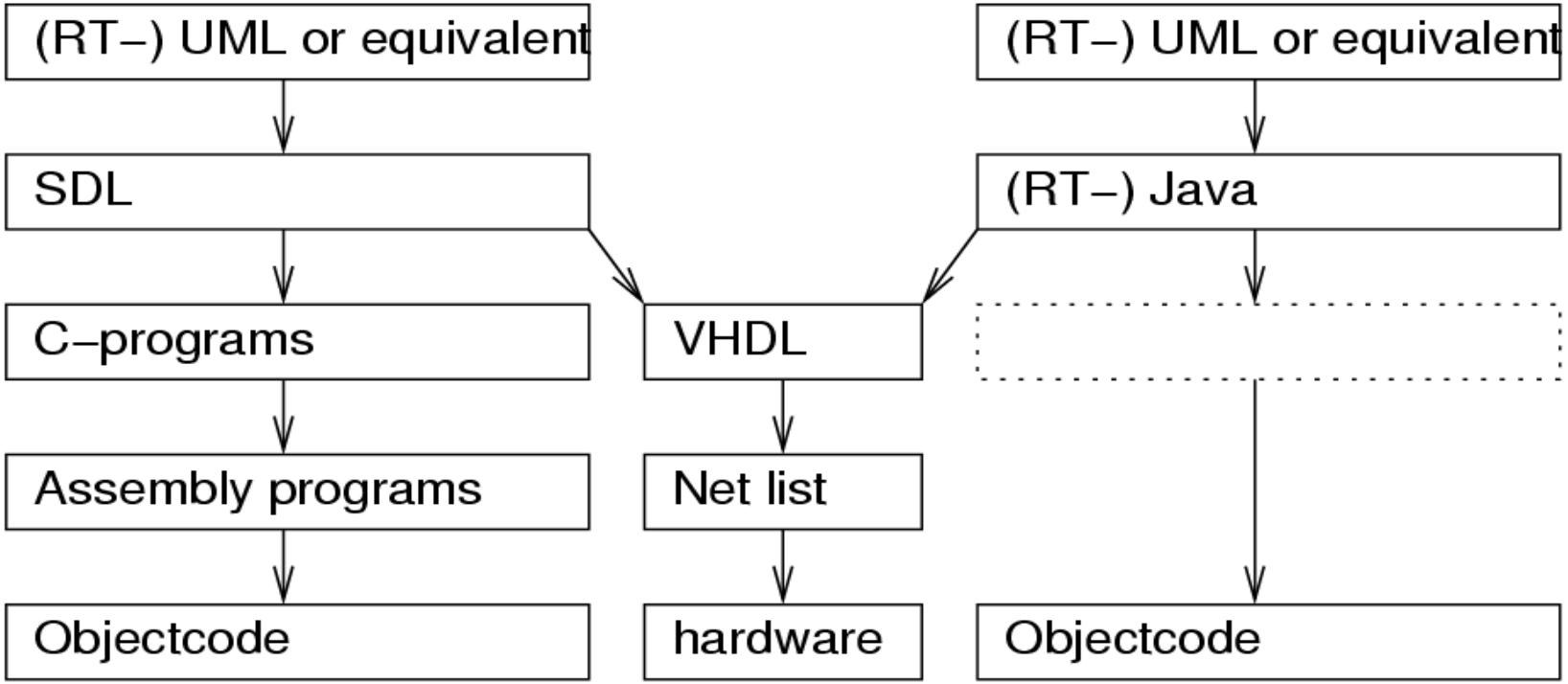
e.g.: StateCharts comprises a static number of processes, nested declaration of processes, and process creation through elaboration in the source.

Language comparison

Language	Behavioral Hierarchy	Structural Hierarchy	Programming Language Elements	Exceptions Supported	Dynamic Process Creation
StateCharts	+	-	-	+	-
VHDL	+	+	+	-	-
SpecCharts	+	-	+	+	-
SDL	+-	+-	+-	-	+
Petri nets	-	-	-	-	+
Java	+	-	+	+	+
SpecC	+	+	+	+	+
SystemC	+	+	+	- (2.0)	- (2.0)
ADA	+	-	+	+	+

How to cope with MoC and language problems in practice?

Mixed approaches:



Transformations between models

- Transformations between models are possible, e.g.
  - Frequent transformation into sequential code
  - Transformations between restricted Petri nets and SDF
  - Transformations between VHDL and C
- Nevertheless, it is best to specify in the most convenient model
- Transformations should be based on the precise description of the semantics
  (e.g. Chen, Sztipanovits et al., DATE, 2007)
  (☞ an advanced course would be nice)

Mixing models of computation: Ptolemy

Ptolemy (UC Berkeley) is an environment for simulating multiple models of computation.

https://ptolemy.berkeley.edu/

Available examples are restricted to a subset of the supported models of computation.

Newton‘s cradle

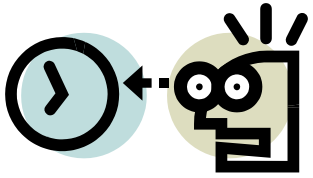
Mixing MoCs: Ptolemy
(Focus on executable models; “mature” models only)

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous
Undefined components		
Communicating finite state machines	FSM, synchronous/reactive MoC	
Data flow		Kahn networks, SDF, dynamic dataflow, discrete time
Petri nets		
Discrete event (DE) model	DE	Experimental distributed DE
Von Neumann model		CSP |
Wireless	Special model for wireless communication	
Continuous time	Partial differential equations	

Mixing models of computation: UML
(Focus on support of early design phases)

Communication/ local computations	**Shared memory**	**Message passing** Synchronous | Asynchronous
Undefined components	*use cases* | *sequence charts, timing diagrams*	
Communicating finite state machines	State diagrams	
Data flow	(Not useful)	Data flow
Petri nets		activity charts
Discrete event (DE) model	-	-
Von Neumann model	-	-

UML for embedded systems?



Initially not designed for real-time.

Initially lacking features:

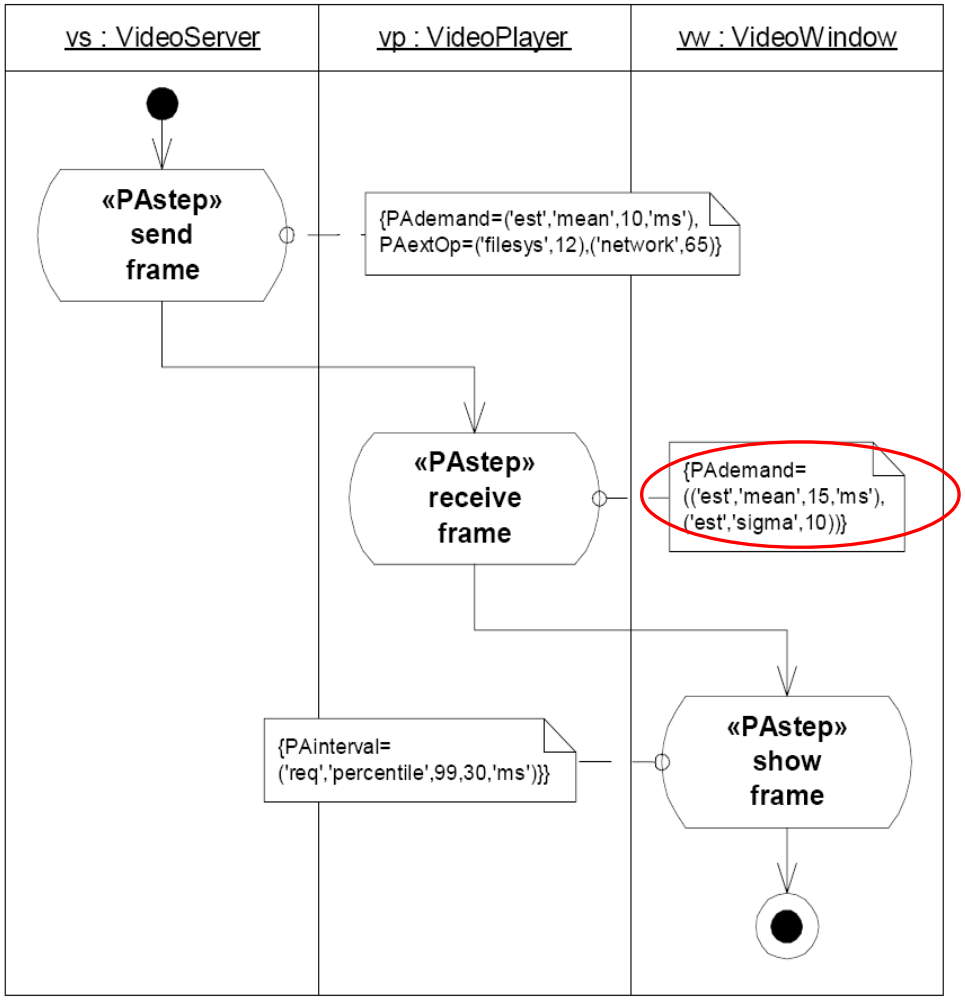
- Partitioning of software into tasks and processes
- specifying timing
- specification of hardware components

Projects on defining profiles for embedded/real-time systems

- Schedulability, Performance and Timing Analysis
- SysML (System Modeling Language)
- UML Profile for SoC
- Modeling and Analysis of Real-Time Embedded Systems
- UML/SystemC, …

Profiles may be incompatible

Example: Activity diagram with annotations



http://www.omg.org/cgi-bin/doc?ptc/2002-03-02

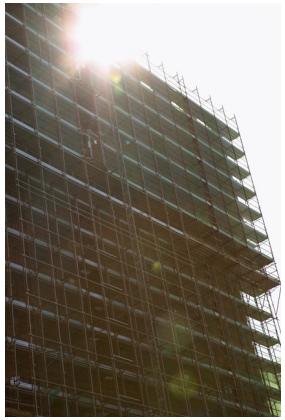
Figure 8-10 Details of the "send video" subactivity with performance annotations

See also W. Müller et al.: UML for SoC

Modeling levels

Levels, at which modeling can be done:

- System level
- Algorithmic level: just the algorithm
- Processor/memory/switch (PMS) level
- Instruction set architecture (ISA) level: function only
- Transaction level modeling (TML): memory reads & writes are just “transactions“ (not cycle accurate)
- Register-transfer level: registers, muxes, adders, … (cycle accurate, bit accurate)
- Gate-level: gates
- Layout level



Tradeoff between accuracy and simulation speed

Example: System level

- Term not clearly defined.
- Here: denotes the entire cyber-physical/embedded system, system into which information processing is embedded, and possibly also the environment.
- Models may include mechanics + information processing. May be difficult to find appropriate simulators. Solutions: VHDL-AMS, SystemC or MATLAB. MATLAB+VHDL-AMS support partial differential equations.
- Challenge to model information processing parts of the system such that the simulation model can be used for the synthesis of the embedded system.

Example: Algorithmic level

- Simulating the algorithms envisioned for the cyber-physical/embedded system.
- No reference to processors or instruction sets.
- Data types may still allow a higher precision than the final implementation.
- Model is **bit-true** if data types selected such that every bit corresponds to exactly one bit in the final implementation
- Single process or sets of cooperating processes.

Segment of MPEG-4

```
for (z=0; z<20; z++)
 for (x=0; x<36; x++) {x1=4*x;
  for (y=0; y<49; y++) {y1=4*y;
   for (k=0; k<9; k++) {x2=x1+k-4;
    for (l=0; l<9; ) {y2=y1+l-4;
     for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
      for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
       if (x3<0 || 35<x3||y3<0||48<y3)
        then_block_1; else else_block_1;
       if (x4<0|| 35<x4||y4<0||48<y4)
        then_block_2; else else_block_2;
}}}}}}
```

Instruction set architecture (ISA)

Algorithms already compiled for the ISA.

Model allows counting the executed # of instructions.

Assembler (MIPS)	Simulated semantics
`and $1,$2,$3`	`Reg[1]:=Reg[2] ^ Reg[3]`
`or $1,$2,$3`	`Reg[1]:=Reg[2] v Reg[3]`
`andi $1,$2,100`	`Reg[1]:=Reg[2] ^ 100`

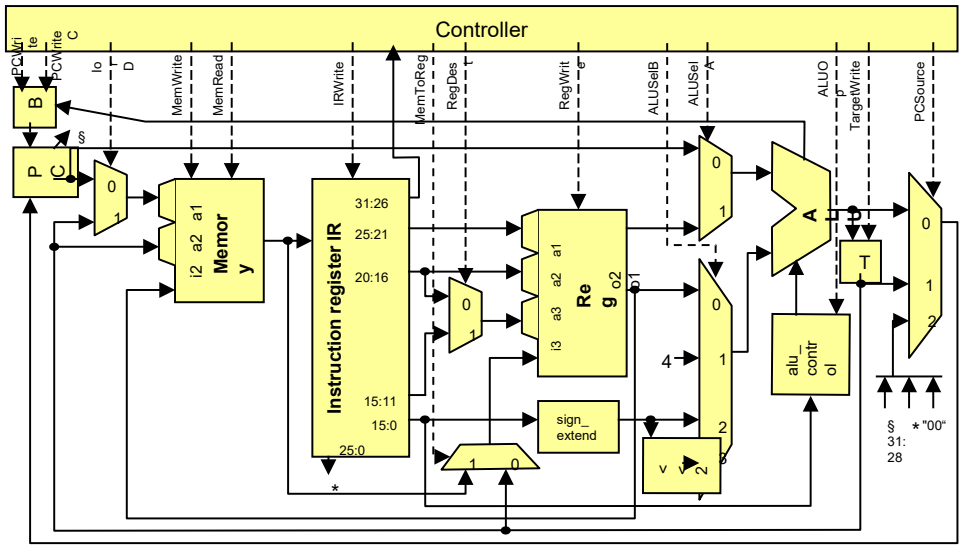
Variations:

- Simulating only of the effect of instructions
- **Transaction-level modeling**: each read/write is one transaction, instead of a set of signal assignments
- **Cycle-true simulations**: exact number of cycles
- **Bit-true simulations:** simulations using exactly the correct number of bits

Example: Register transfer level (RTL)

Modeling of all components at the register-transfer level, including

- arithmetic/logic units (ALUs),
- registers,
- memories,
- muxes and
- decoders.

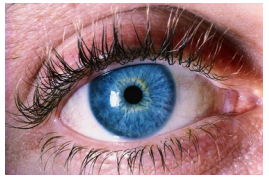


Models at this level are always cycle-true.

Automatic synthesis from such models is frequently feasible.

What‘s the bottom line?

- The prevailing technique for writing embedded SW has inherent problems; some of the difficulties of writing embedded SW are not resulting from design constraints, but from the modeling.
- However, there is no ideal modeling technique.
- The choice of the technique depends on the application.
- Check code generation from non-imperative models
- There is a tradeoff between the power of a modeling technique and its analyzability.
- It may be necessary to combine modeling techniques.
- **In any case, open your eyes & think about the model before you write down your spec! Be aware of pitfalls.**



- You may be forced, to use imperative models, but you can still implement, for example, finite state machines or KPNs in Java.

Summary

- Imperative Von-Neumann models
  - Problems resulting from access to shared resources and mutual exclusion (e.g. potential deadlock)
  - Communication built-in or by libraries
- Comparison of models
  - Expressiveness vs. analyzability
  - Process creation
  - Mixing models of computation
    - Ptolemy & UML
    - Using FSM and KPN models in imperative languages, etc.

technische universität
dortmund

fakultät für informatik
informatik 12

Embedded System Hardware

Peter Marwedel
TU Dortmund,
Informatik 12

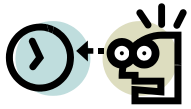
2012 年 11 月 13 日

Motivation

(see lecture 1):
"*The development of ES cannot ignore the underlying HW characteristics. Timing, memory usage, power consumption and physical failures are important*."

Reasons for considering hard- and software:

- Real-time behavior



- Efficiency
  - Energy

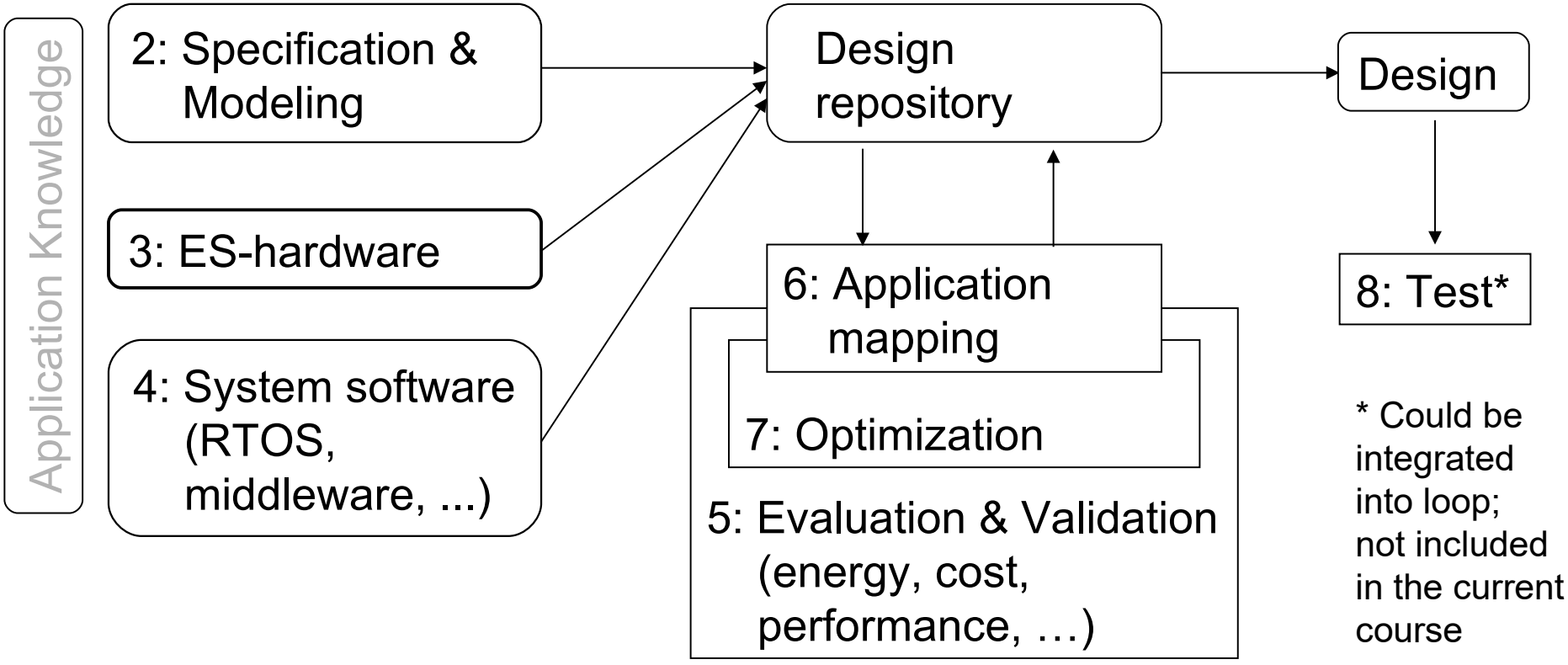
$$\int P\,dt$$

  - …
- Security

- Reliability

- …

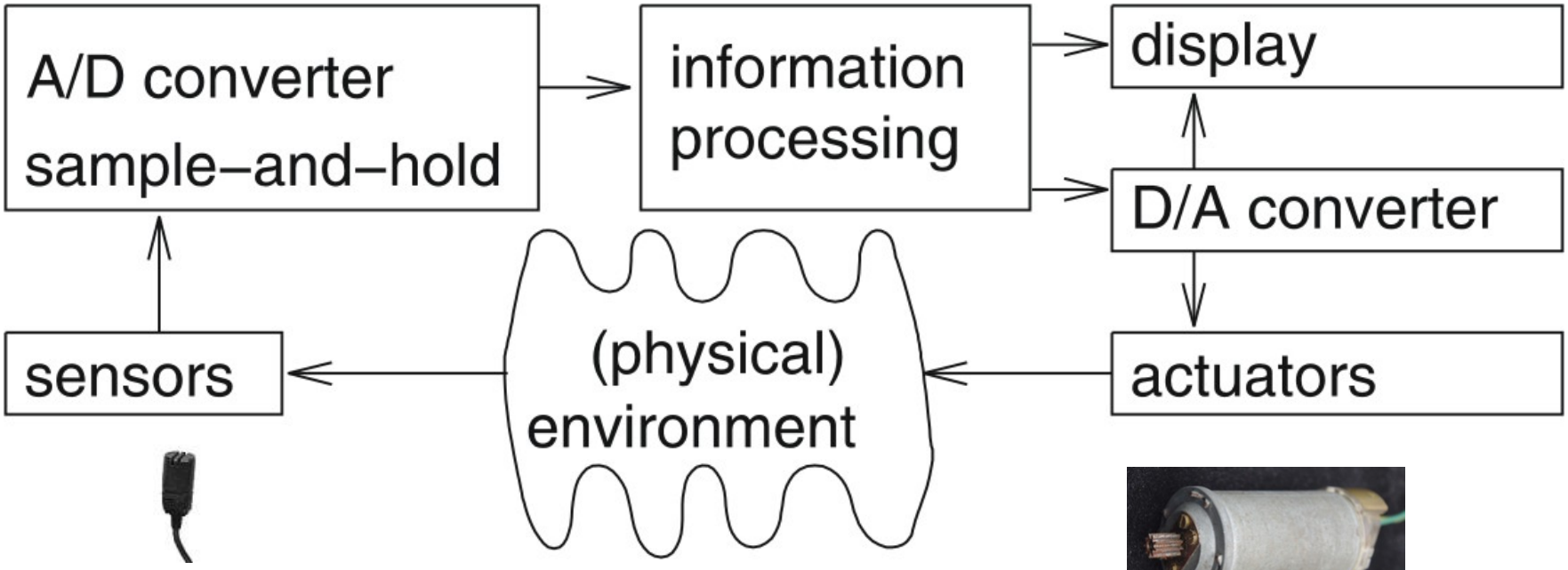
Structure of this course



Generic loop: tool chains differ in the number and type of iterations
Numbers denote sequence of chapters

Embedded System Hardware

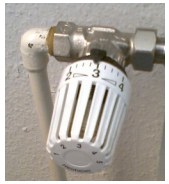
Embedded system hardware is frequently used in a loop (***"hardware in a loop"***):

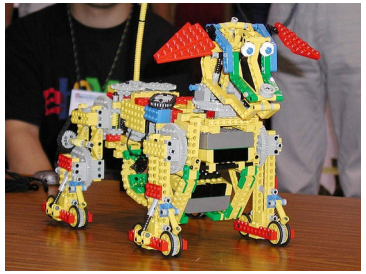


☞ cyber-physical systems

Many examples of such loops

- Heating
- Lights
- Engine control
- Power supply
- …
- Robots





Sensors

Processing of physical data starts with capturing this data. Sensors can be designed for virtually every physical and chemical quantity, including

- weight, velocity, acceleration, electrical current, voltage, temperatures, and
- chemical compounds.

Many physical effects used for constructing sensors.

Examples:

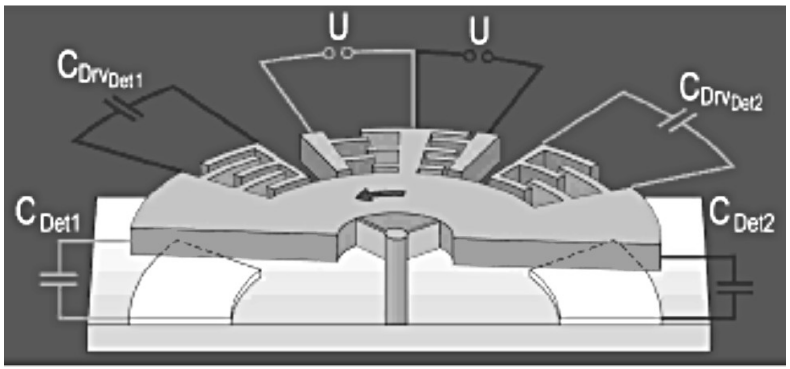
- law of induction (generat. of voltages in a magnetic field),
- light-electric effects.

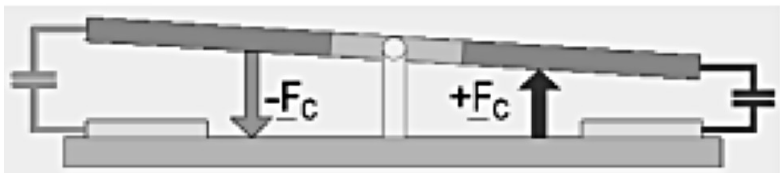
Huge amount of sensors designed in recent years.

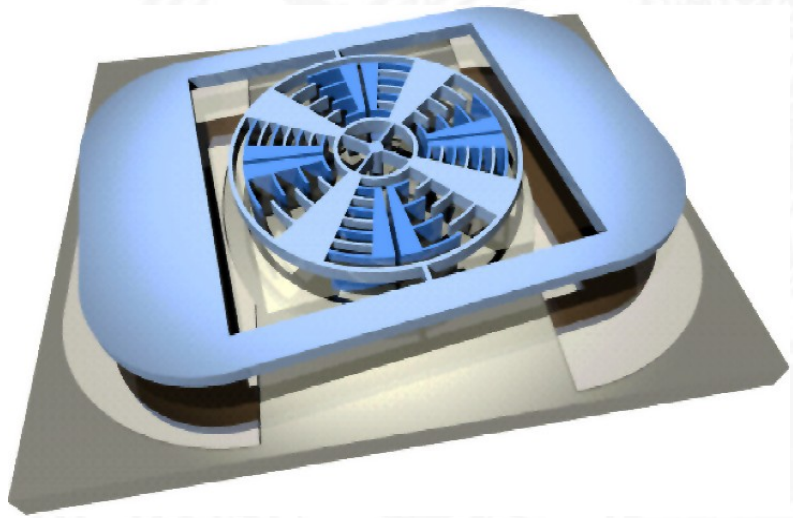
OMM-Beispiele

Drehratensensor

- schwingende Struktur
- Sekundärschwingung durch äußere Drehbewegung
- kapazitive Auswertung

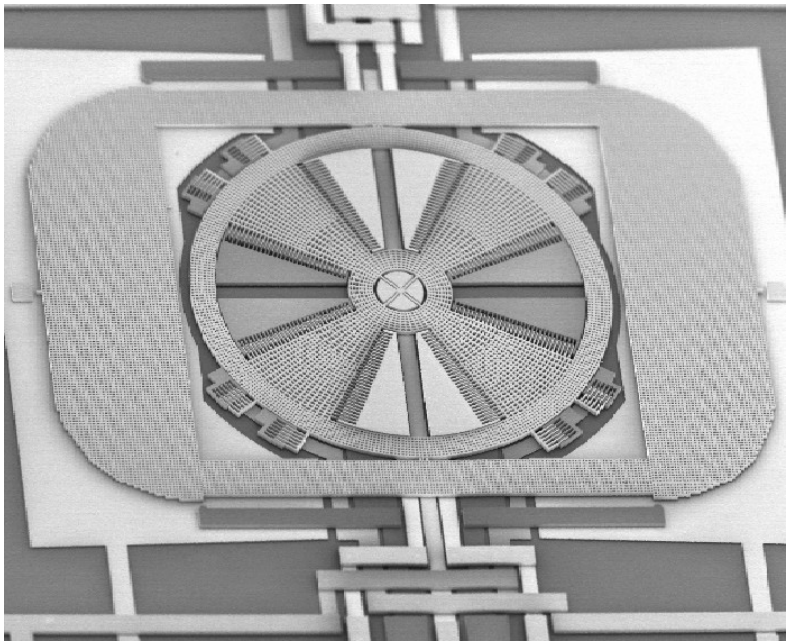


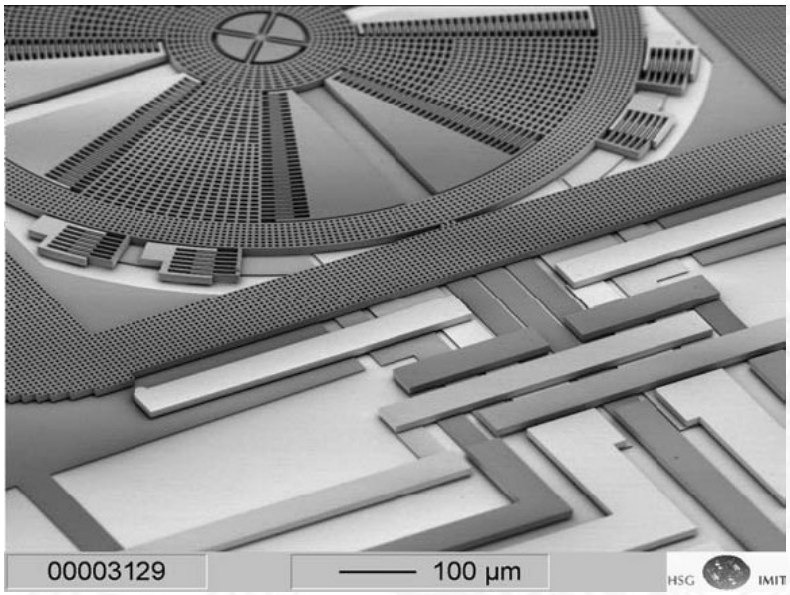




OMM-Beispiele (cont.)

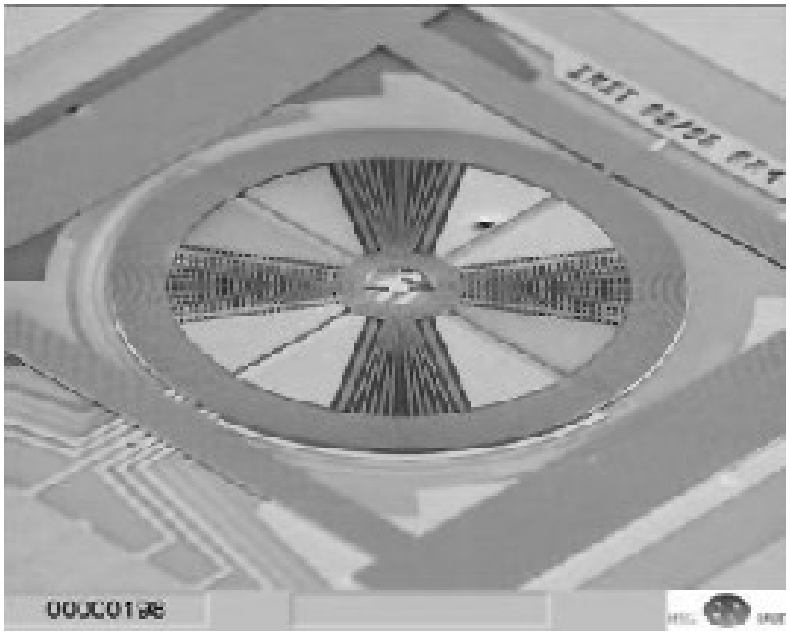
Drehratensensor

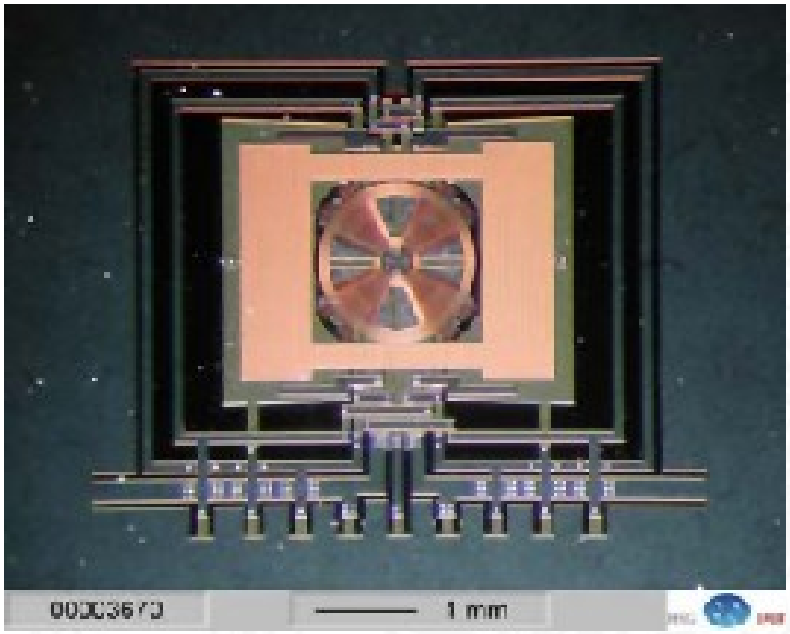




OMM-Beispiele (cont.)

Drehratensensor

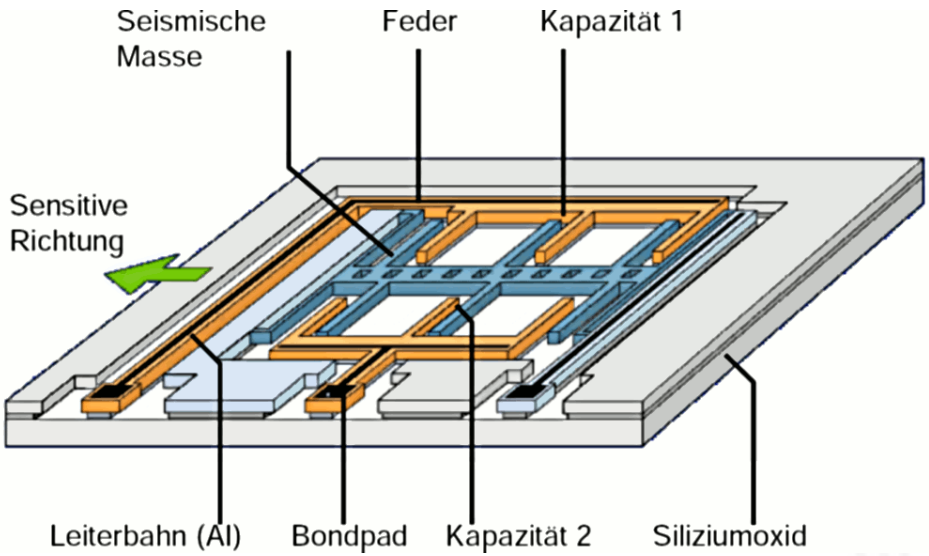


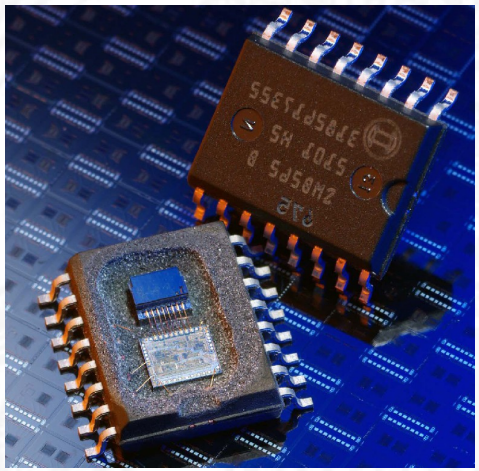


OMM-Beispiele

Beschleunigungssensor 1

- schwingende Struktur
- Überlagerung bei äußerer Beschleunigung
- kapazitive Auswertung $a \approx \frac{C_1 - C_2}{C_1 + C_2}$





OMM-Beispiele (cont.)

Beschleunigungssensor 1

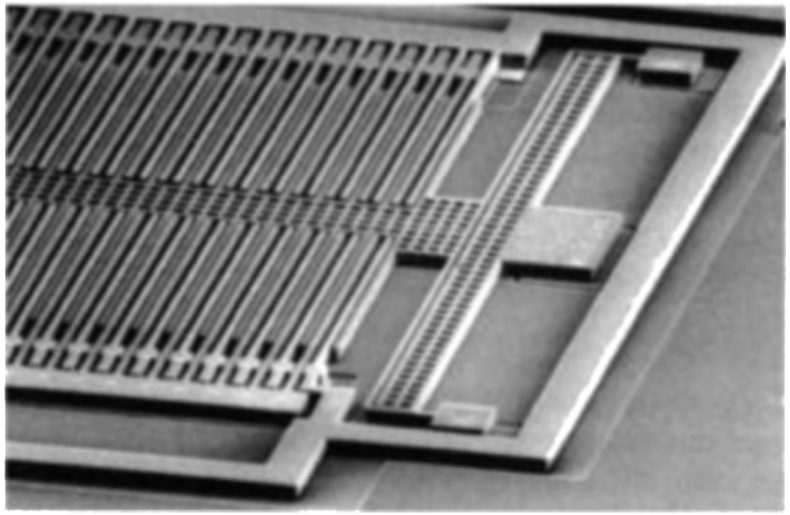
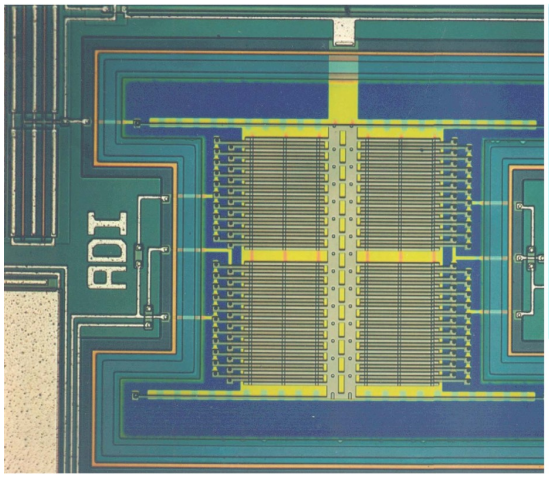
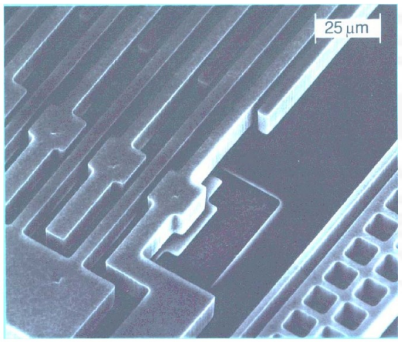


Figure 1: Accelerometer, photo courtesy Bosch

Sensorstruktur



Detail der Elektrodenstruktur

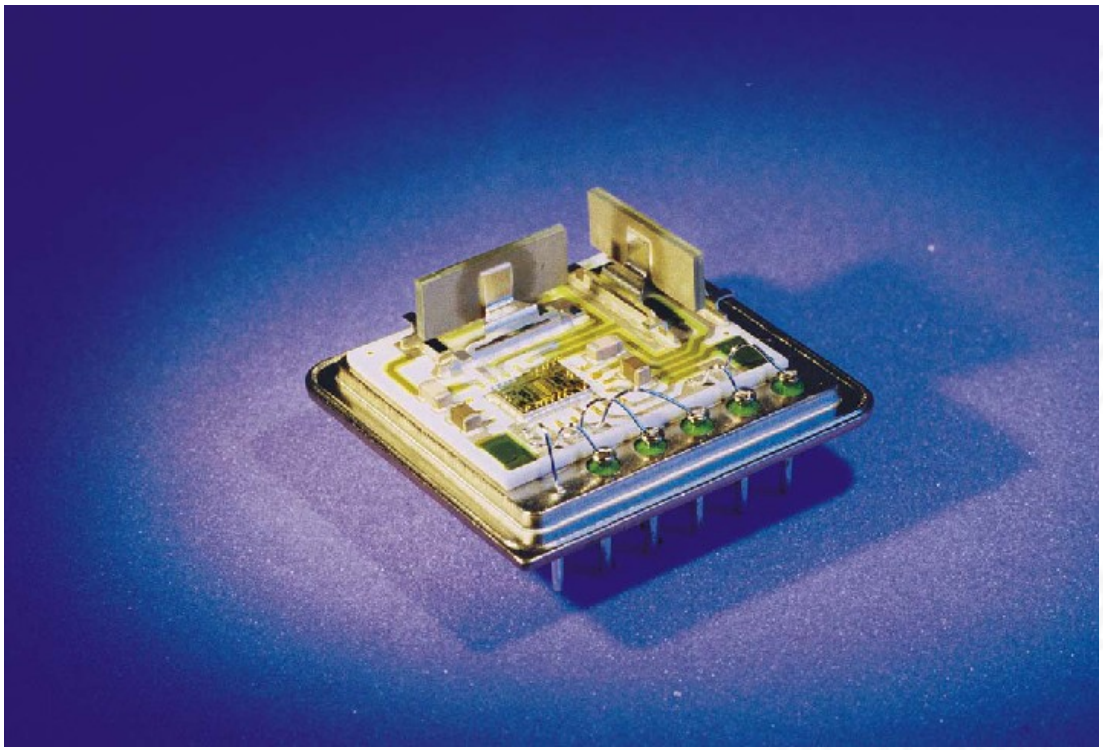


Quelle: Bosch, Motorola

Bulk-Beispiele (cont.)

Beschleunigungssensor

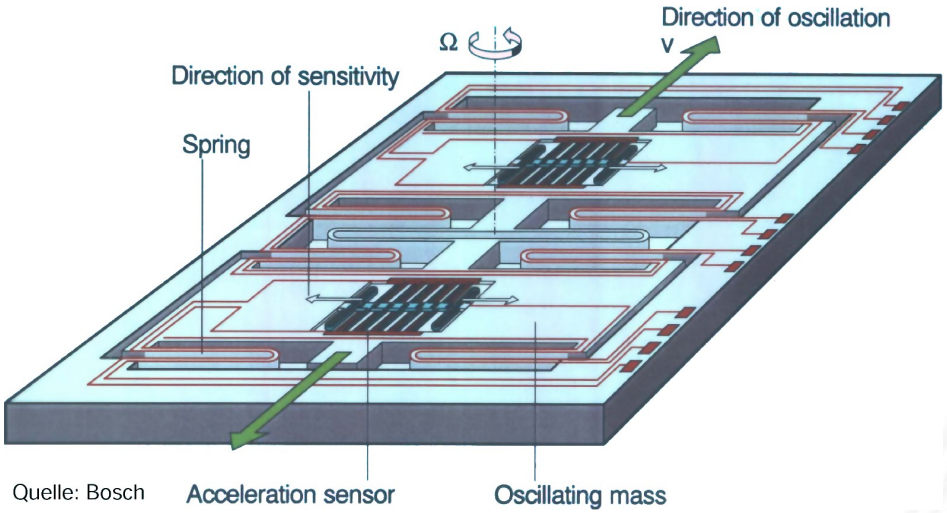
- 3D Beschleunigungssensor



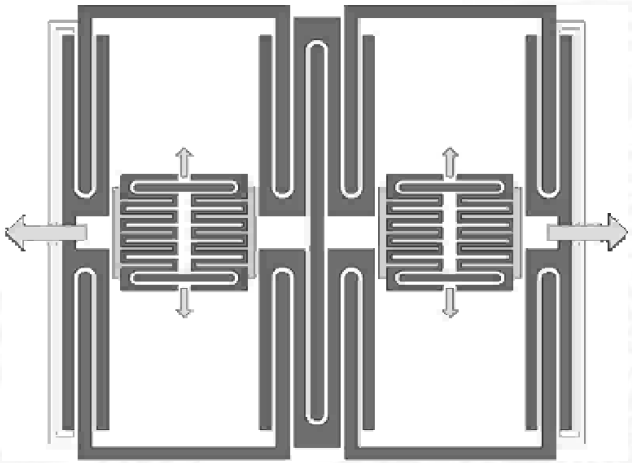
Quelle: Bosch

Bulk-Beispiele

Drehratensensor

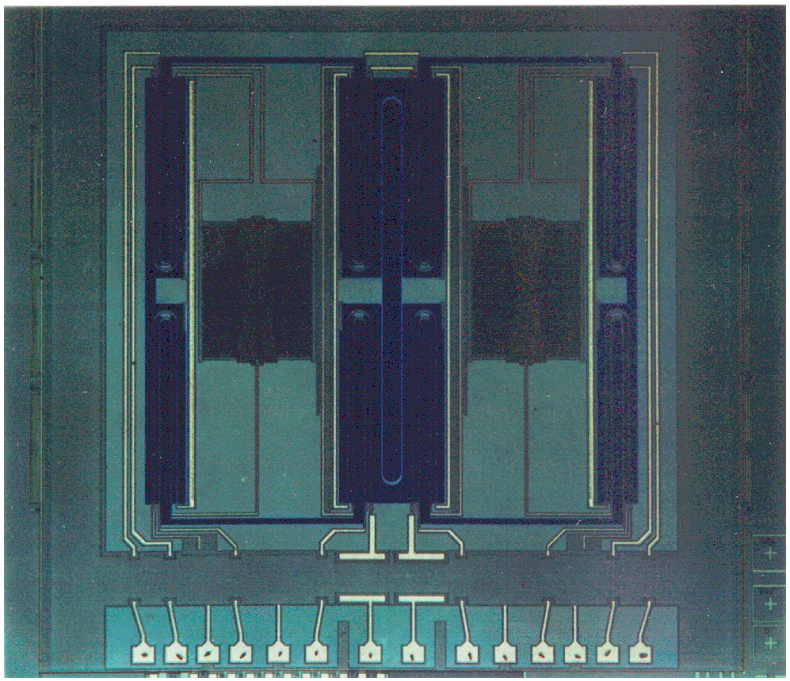


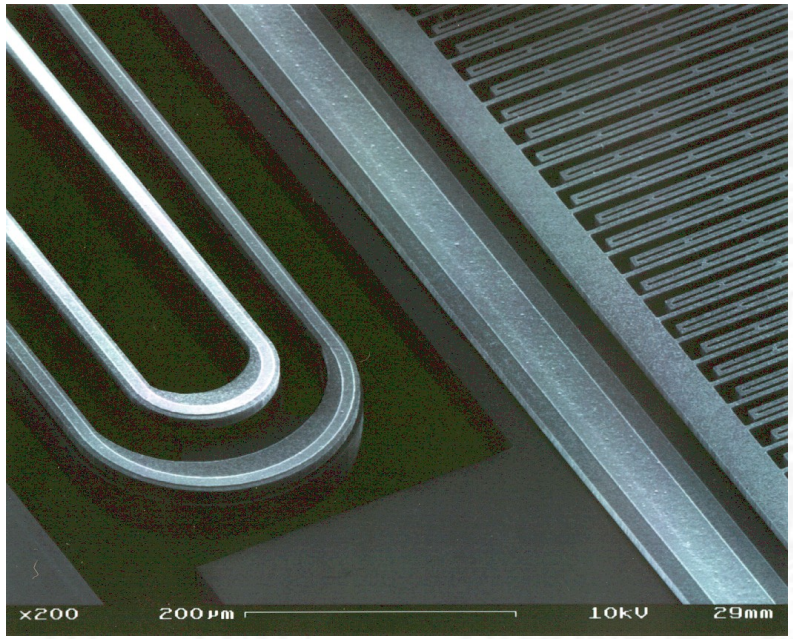
Quelle: Bosch



Bulk-Beispiele (cont.)

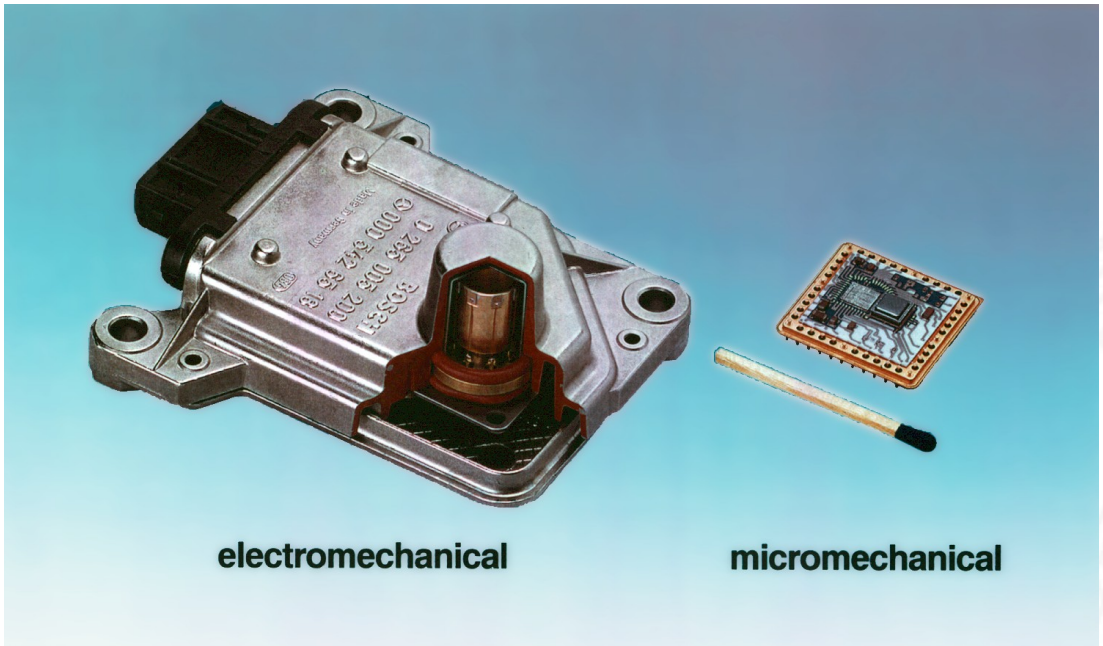
Drehratensensor



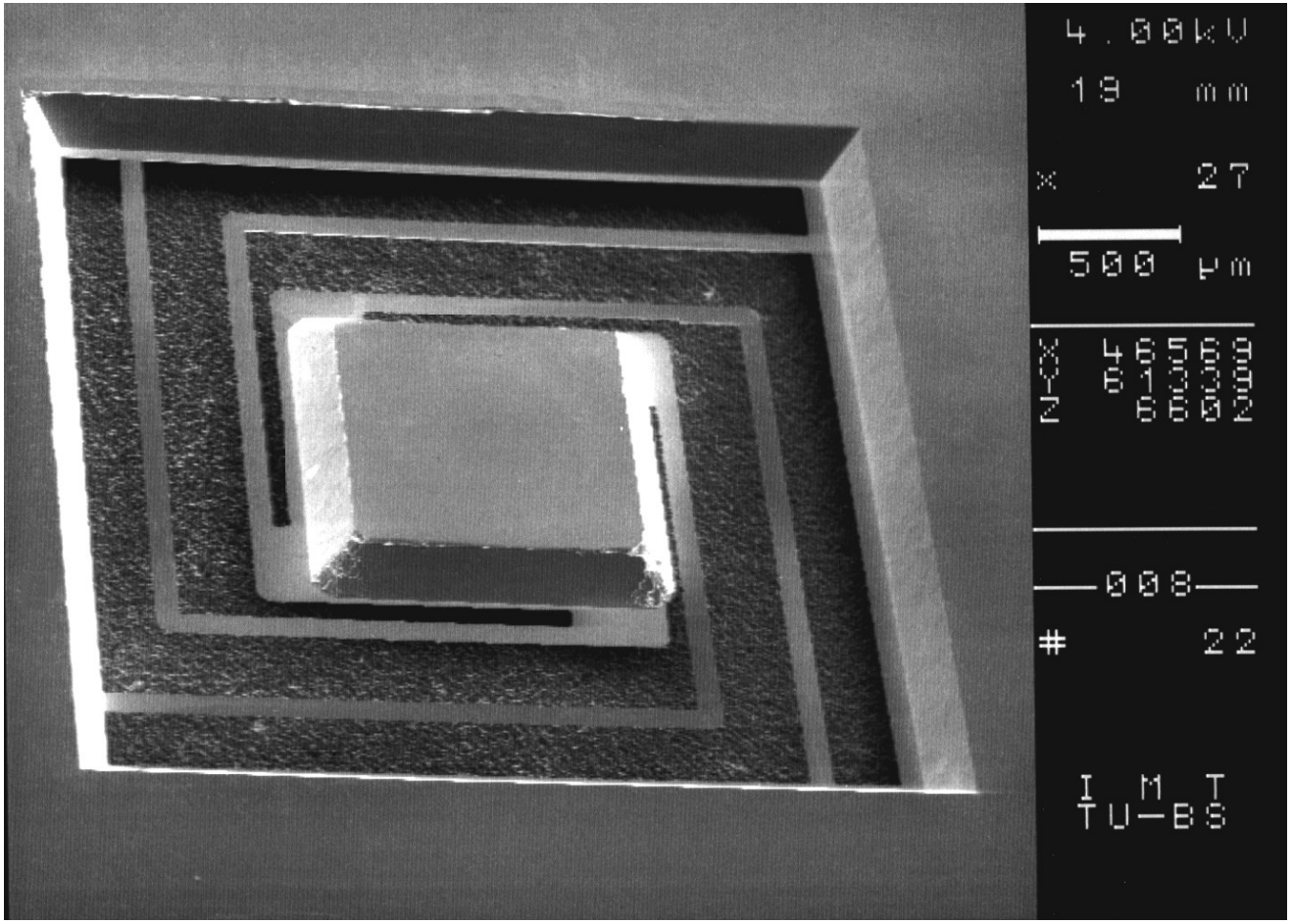


Bulk-Beispiele (cont.)

Drehratensensor



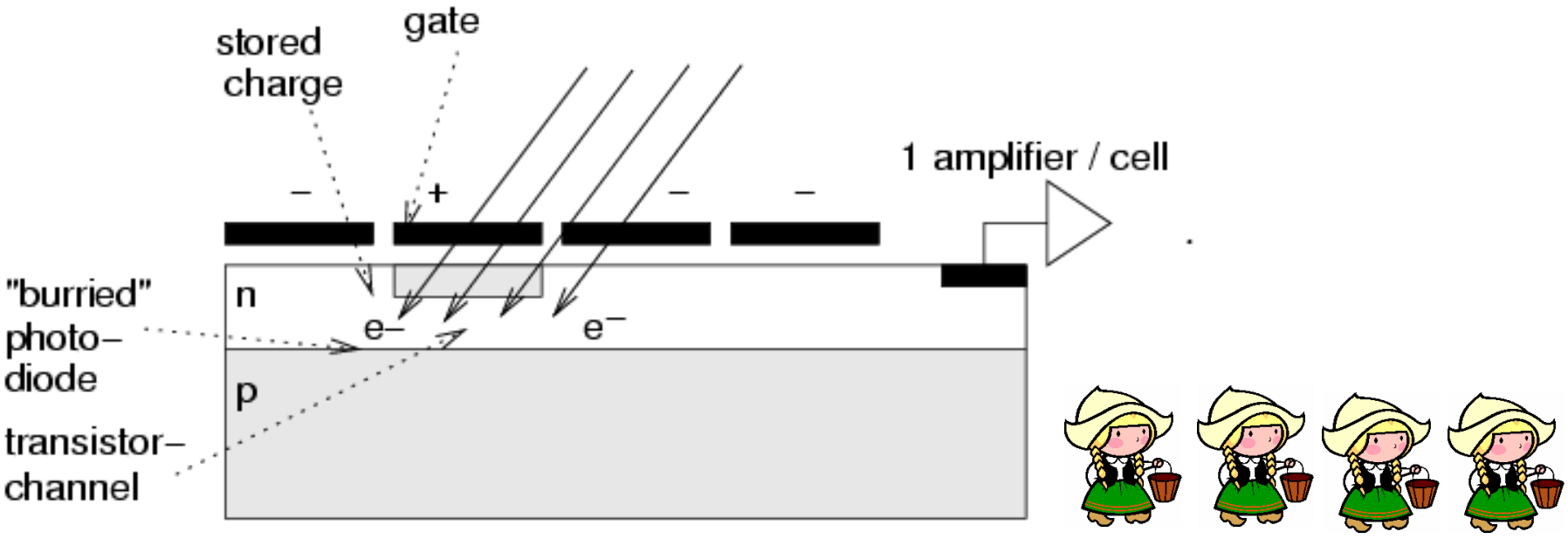
Example: Acceleration Sensor



Courtesy & ©: S. Bütgenbach, TU Braunschweig

Charge-coupled devices (CCD) image sensors

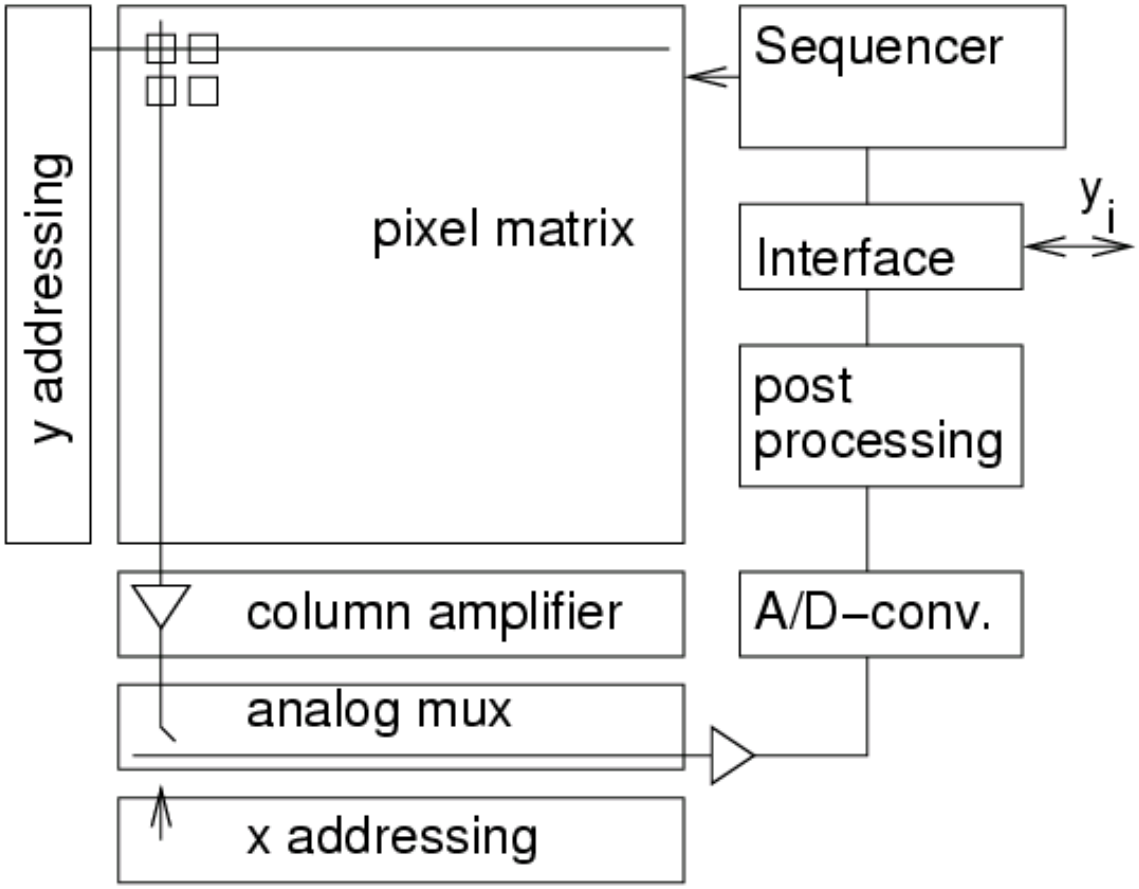
Based on charge transfer to next pixel cell



Corresponding to “bucket brigade device”
(German: “*Eimerkettenschaltung*”)

CMOS image sensors

Based on standard production process for CMOS chips, allows integration with other components.



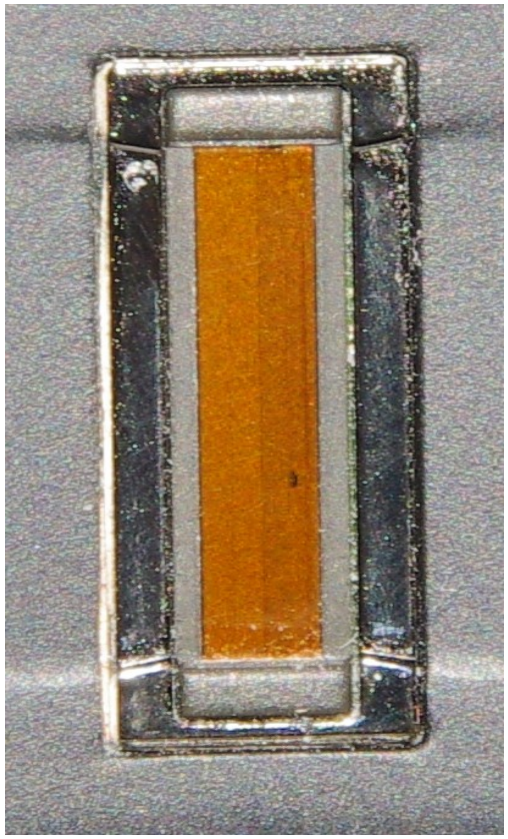
Comparison CCD/CMOS sensors

Property	**CCD**	**CMOS**
Technology optimized for	Optics	VLSI technology
Technology	Special	Standard
Smart sensors	No, no logic on chip	Logic elements on chip
Access	Serial	Random
Size	Limited	Can be large
Power consumption	Low	Larger
Video mode	Possibly too slow	ok
Applications	Situation is changing over the years	

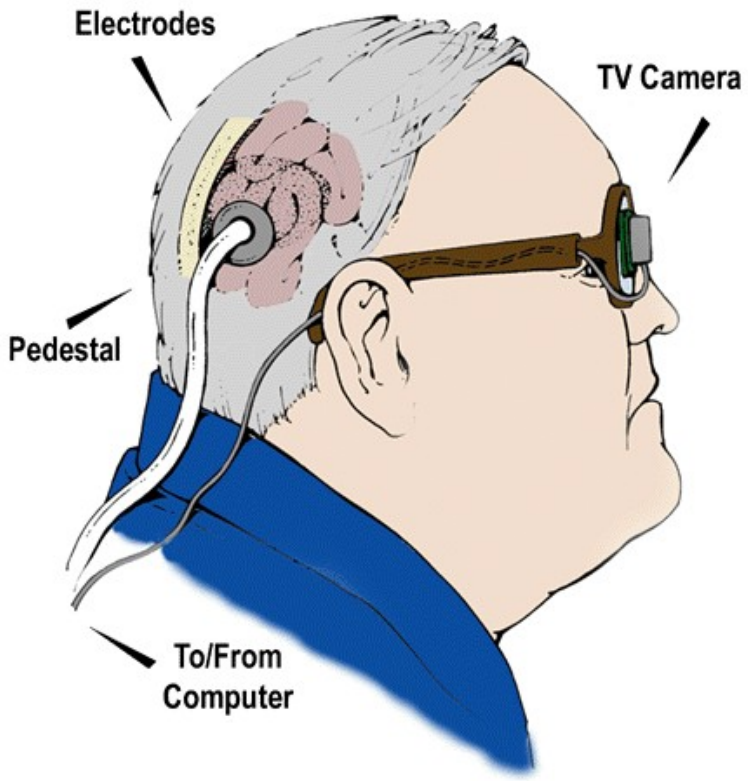
See also B. Dierickx: CMOS image sensor concepts. Photonics West 2000 Short course (Web)

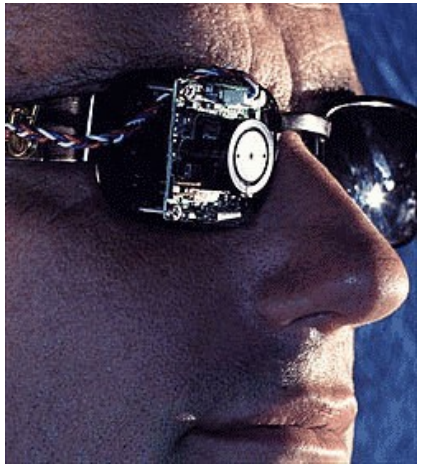
Example: Biometrical Sensors

e.g.: Fingerprint sensor



Artificial eyes (1)





© Dobelle Institute (was at www.dobelle.com)

Artificial eyes (2)

- Translation into sound
  https://www.seeingwithsound.com/etumble.htm

Other sensors

- Rain sensors for wiper control
  (“Sensors multiply like rabbits“ [ITT automotive])
- Pressure sensors
- Proximity sensors
- Engine control sensors
- Hall effect sensors

Signals

Sensors generate *signals*

Definition: a **signal** s is a mapping
from the time domain D_T to a value domain D_V:

$$s : D_T \rightarrow D_V$$

D_T : continuous or discrete time domain

D_V : continuous or discrete value domain.

Discretization

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 11 月 13 日

Discretization of time

Digital computers require discrete sequences of physical values

$$s : D_T \rightarrow D_V$$

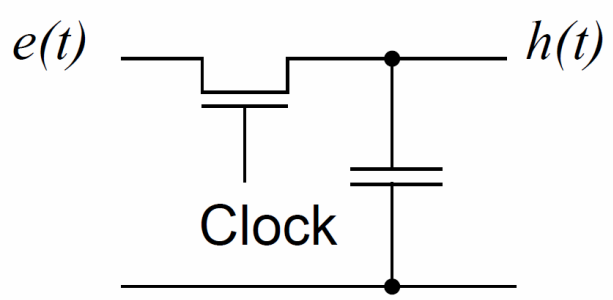
Discrete time domain

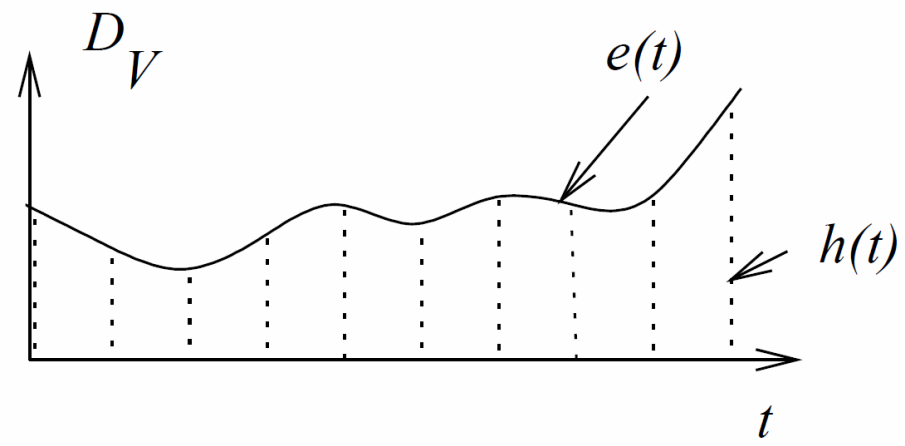
☞ Sample-and-hold circuits

Sample-and-hold circuits

Clocked transistor + capacitor;

Capacitor stores sequence values





$e(t)$ is a mapping $\mathbb{R} \rightarrow \mathbb{R}$

$h(t)$ is a **sequence** of values or a mapping $\mathbb{Z} \rightarrow \mathbb{R}$

Do we lose information due to sampling?

Would we be able to reconstruct input signals from the sampled signals?

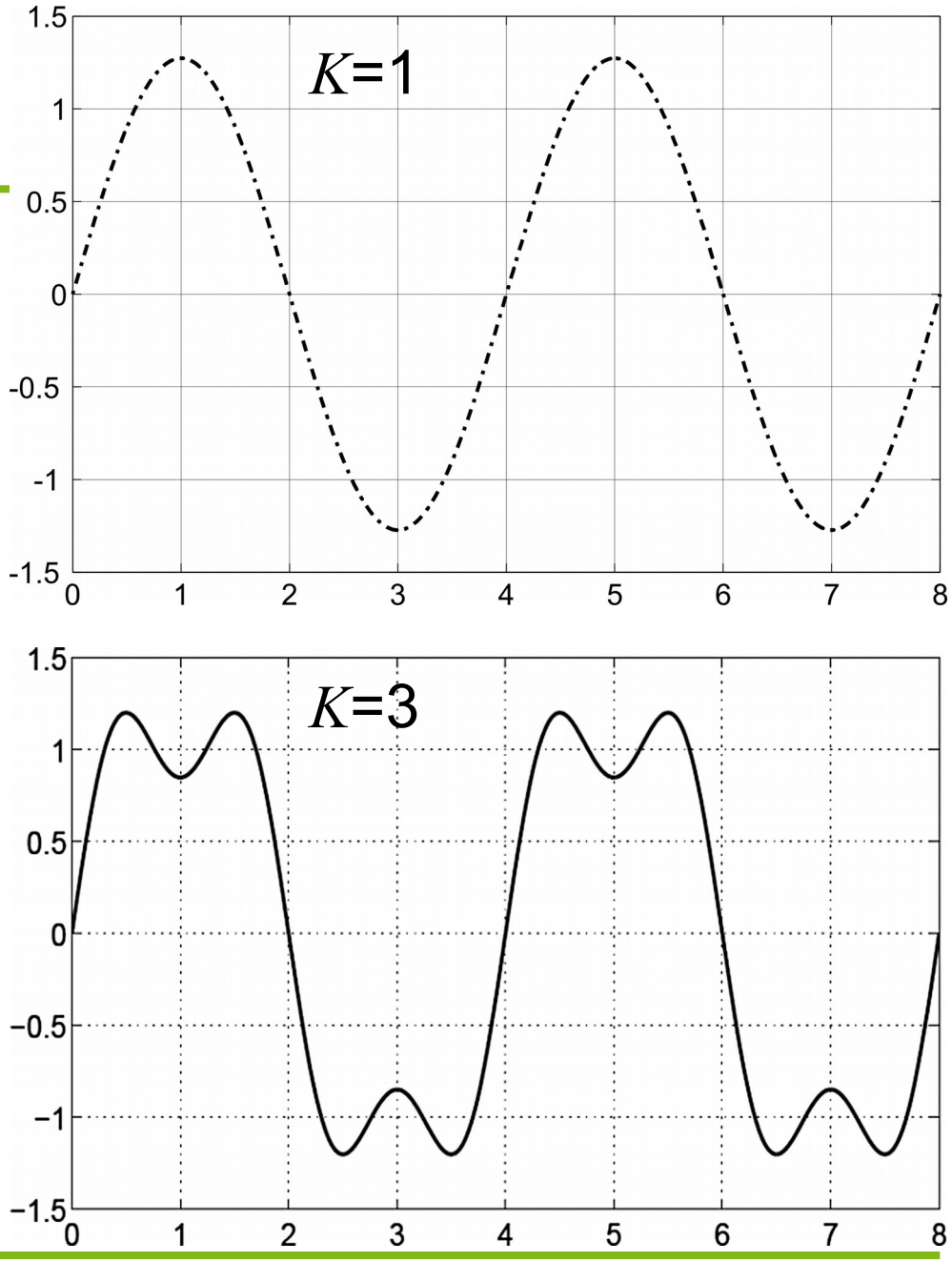
☞ approximation of signals by sine waves.

Approximation of a square wave (1)

Target: square wave with period $T=4$

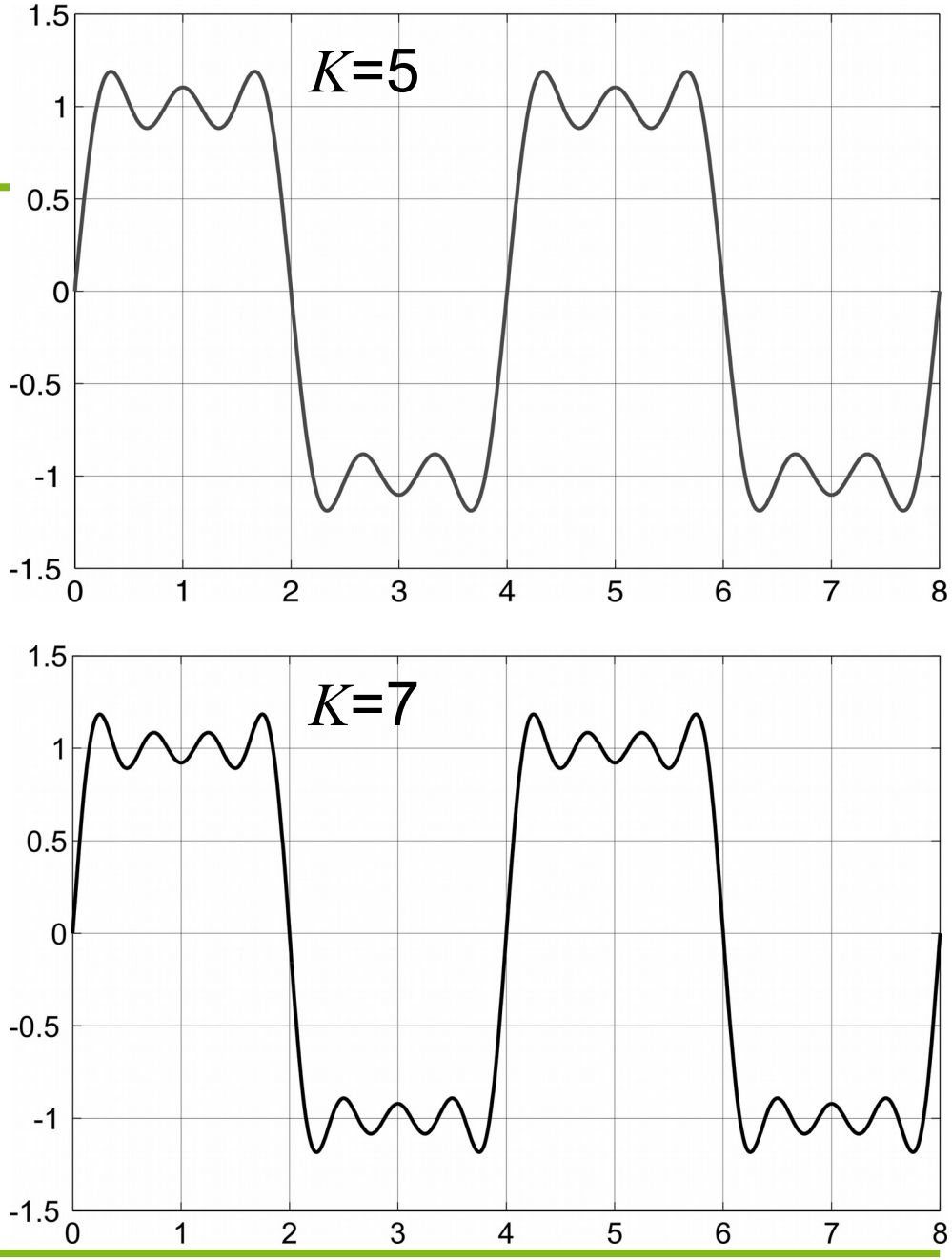
$$e'_K(t) = \sum_{k=1,3,5,..}^{K} \frac{4}{\pi k} \sin\left(\frac{2\pi k t}{T}\right)$$

with $\forall k$: $T_k = T/k$: periods of contributions to e'



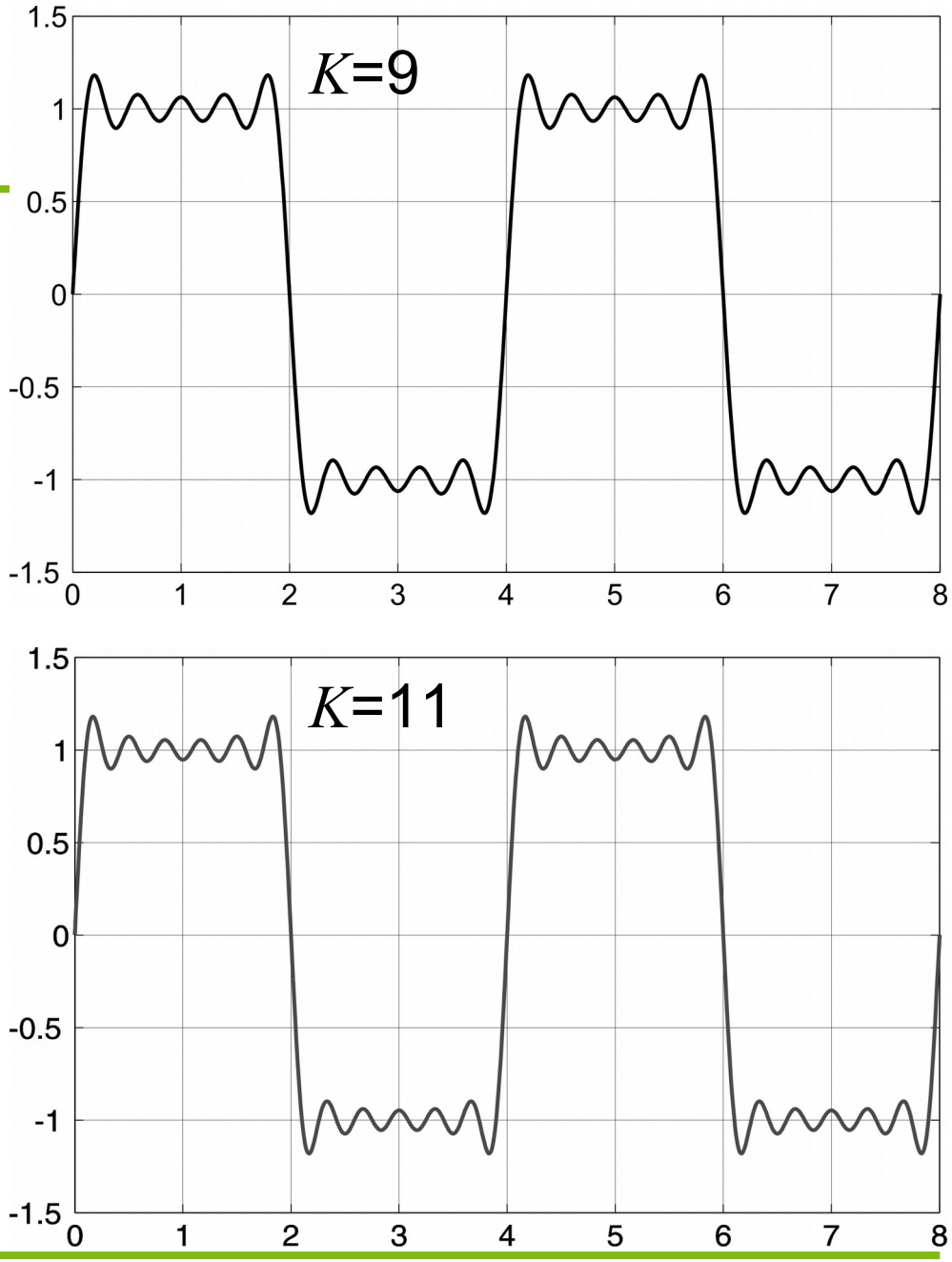
Approximation of a square wave (2)

$$e'_K(t) = \sum_{k=1,3,5,..}^{K} \frac{4}{\pi k} \sin\left(\frac{2\pi k t}{T}\right)$$



Approximation of a square wave (3)

$$e'_K(t) = \sum_{k=1,3,5,..}^{K} \frac{4}{\pi k} \sin\left(\frac{2\pi k t}{T}\right)$$



Linear transformations

Let $e_1(t)$ and $e_2(t)$ be signals

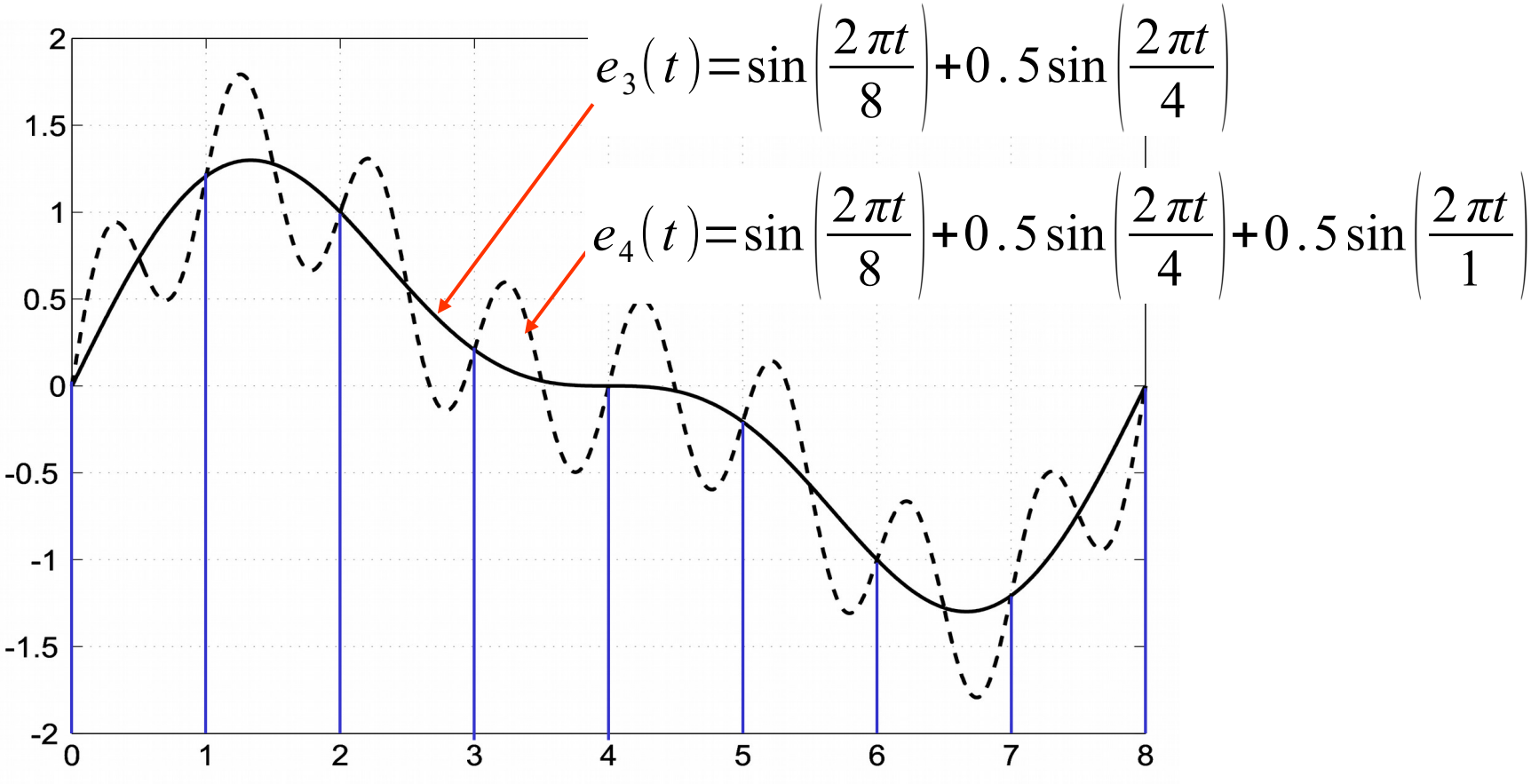
Definition: A transformation Tr of signals is linear iff

$$Tr(e_1 + e_2) = Tr(e_1) + Tr(e_2)$$

In the following, we will consider linear transformations.

☞ We consider sums of sine waves instead of the original signals.

Aliasing (1)



Periods of p=8,4,1
Indistinguishable if sampled at integer times, p_s=1

Aliasing (2)

☞ Reconstruction impossible, if not sampling frequently enough

How frequently do we have to sample?

Nyquist criterion (sampling theory):

Aliasing can be avoided if we restrict the frequencies of the incoming signal to less than half of the sampling rate.

$p_s < ½\, p_N$ where p_N is the period of the "fastest" sine wave

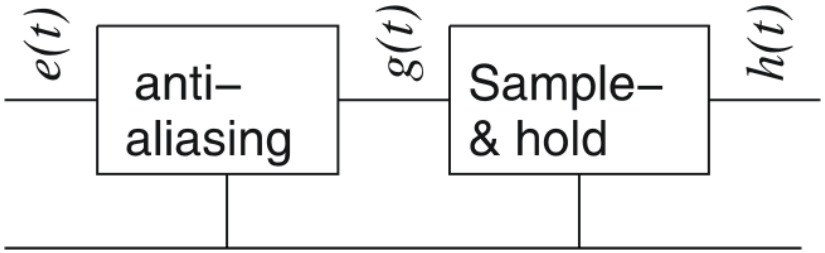
or $f_s > 2\, f_N$ where f_N is the frequency of the "fastest" sine wave

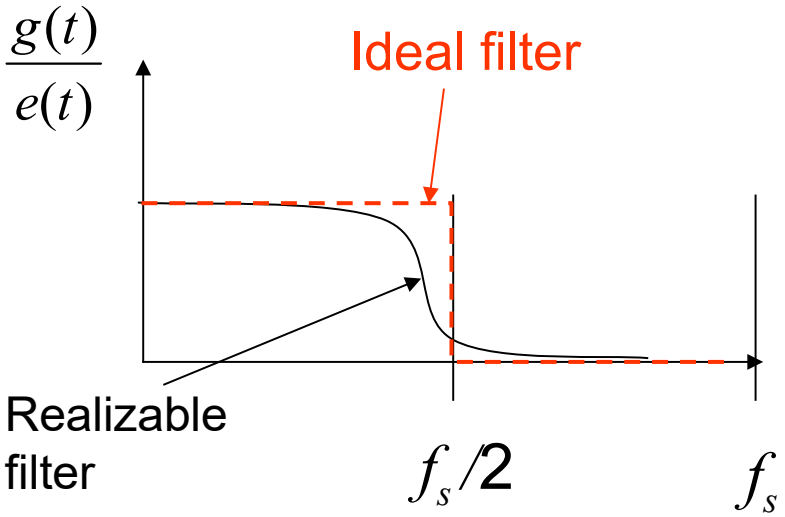
f_N is called the **Nyquist frequency**, f_s is the **sampling rate**.

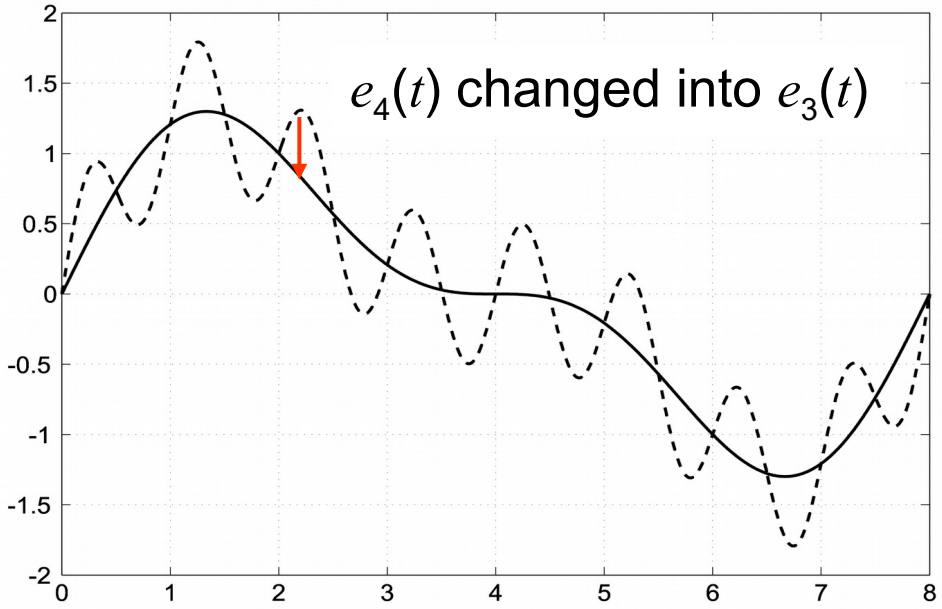
See e.g. [Oppenheim/Schafer, 2009]

Anti-aliasing filter

A filter is needed to remove high frequencies

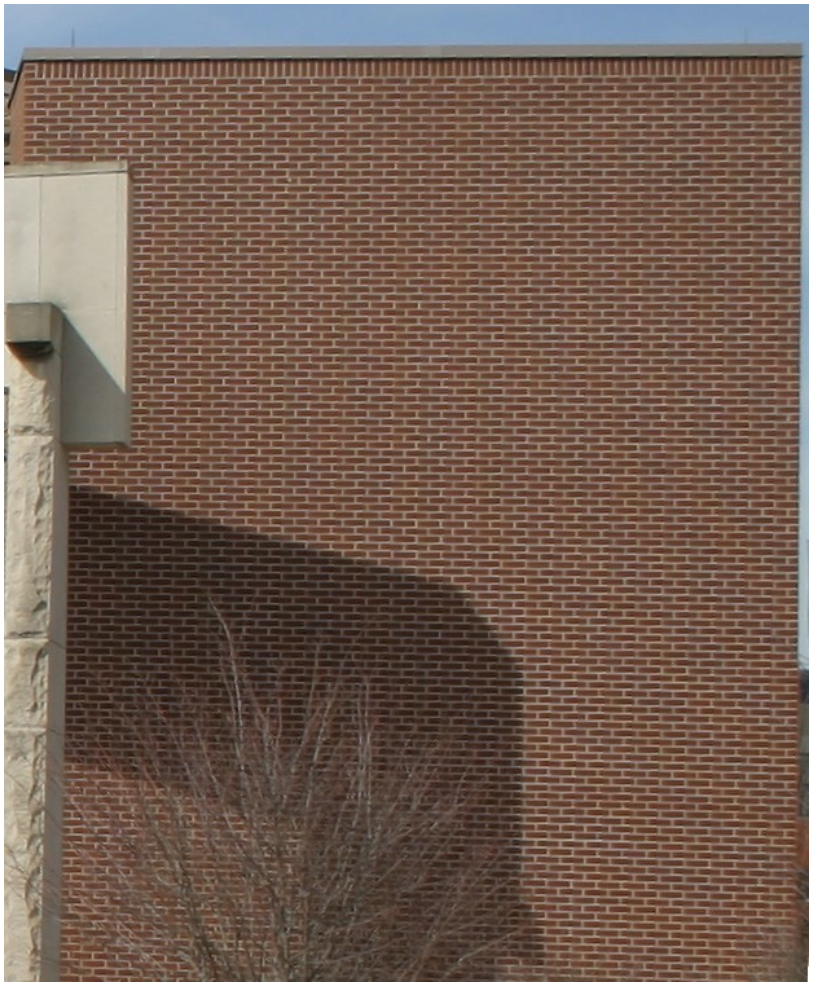




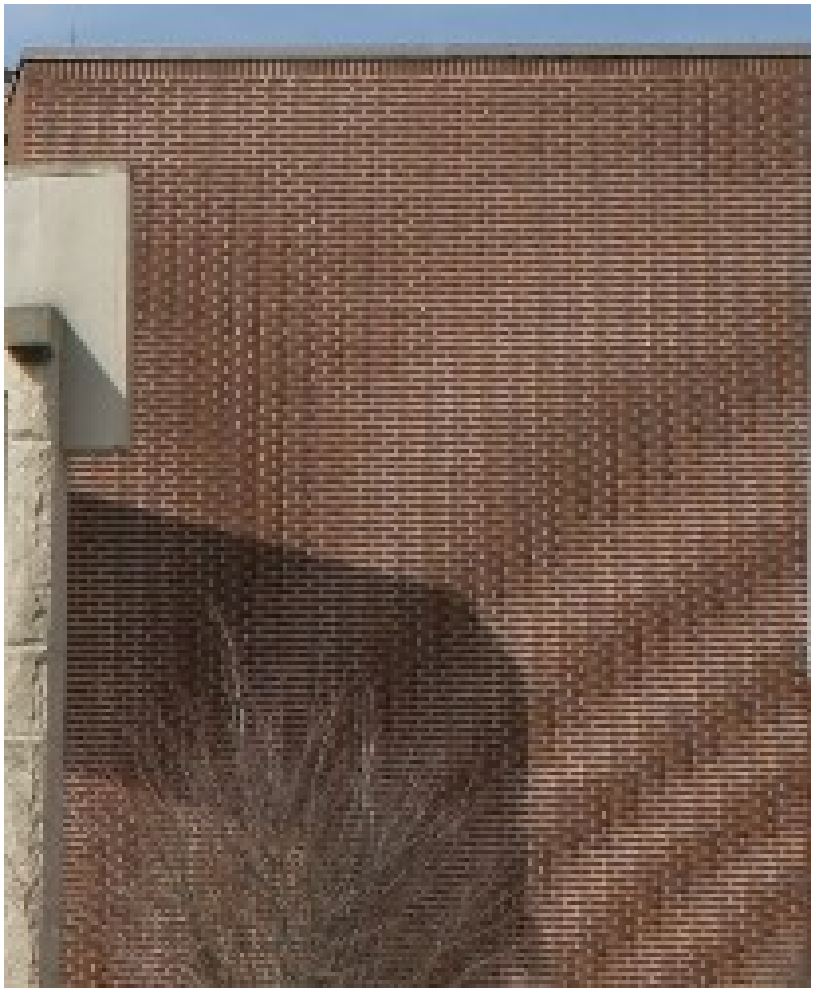


Examples of aliasing in computer graphics (1)

Original

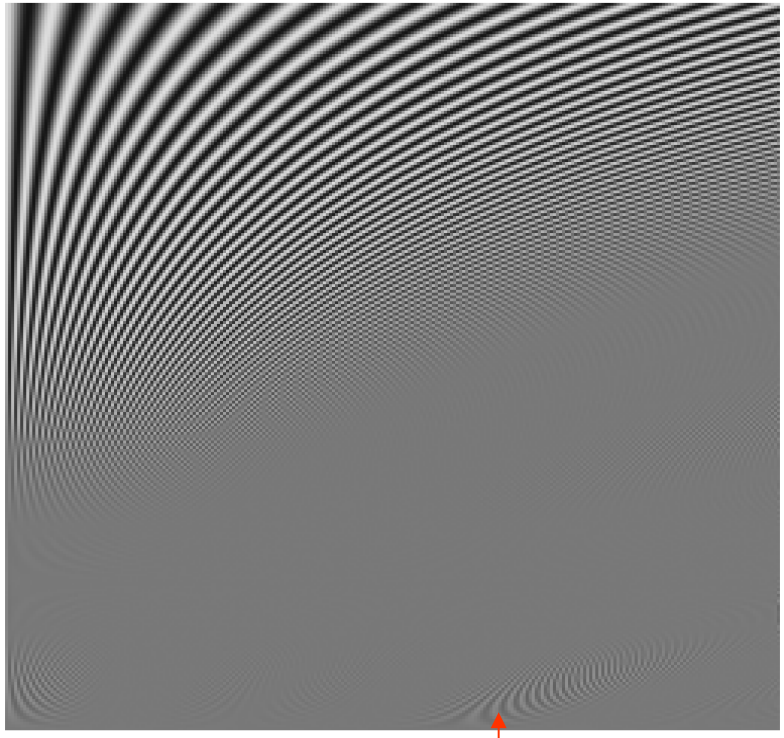


Sub-sampled, no filtering



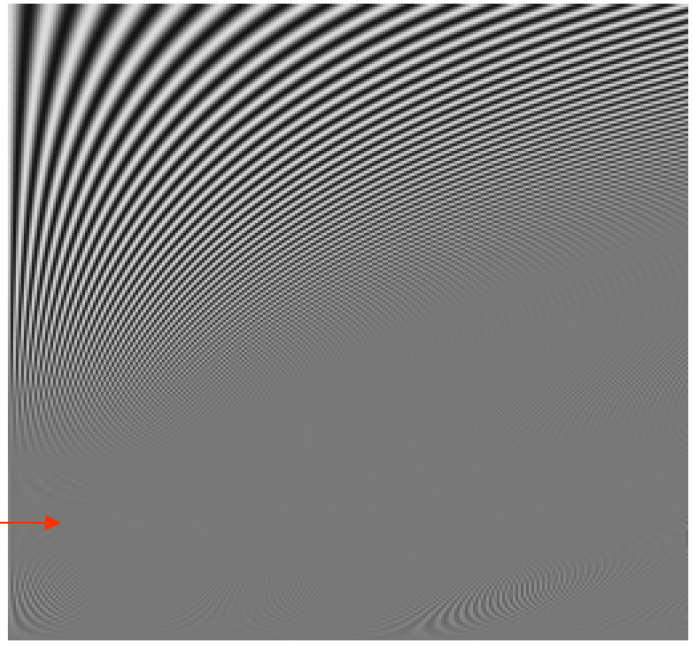
Examples of aliasing in computer graphics (2)

Original (pdf screen copy)

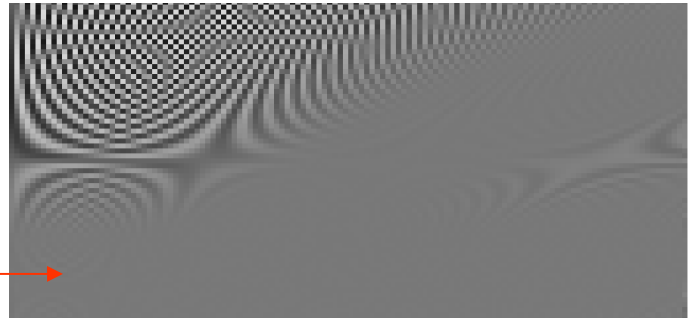


Impact of rasterization

Filtered & sub-sampled



Sub-sampled, no filtering



Discretization of values: A/D-converters

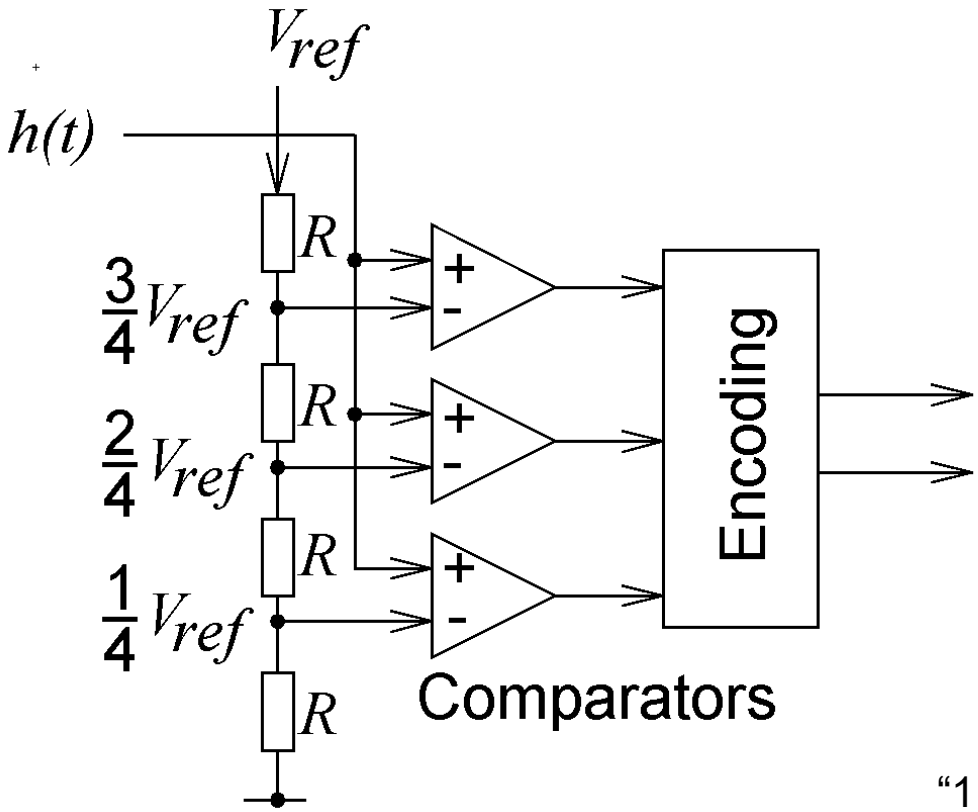
Digital computers require digital form of physical values

$$s\colon D_T \rightarrow D_V$$

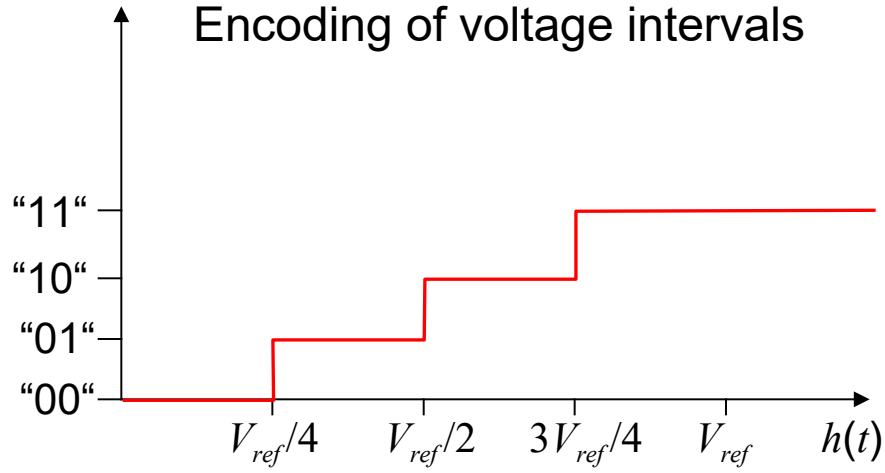
Discrete value domain

☞ A/D-conversion; many methods with different speeds.

Flash A/D converter



No decoding of $h(t) > V_{ref}$



Resolution

- Resolution (in bits): number of bits produced
- Resolution Q (in volts): difference between two input voltages causing the output to be incremented by 1

$$Q = \frac{V_{FSR}}{n}$$ with

Q : resolution in volts per step

V_{FSR} : difference between largest and smallest voltage

n : number of voltage intervals

Example:
$Q = V_{ref}/4$ for the previous slide

Resolution and speed of Flash A/D-converter

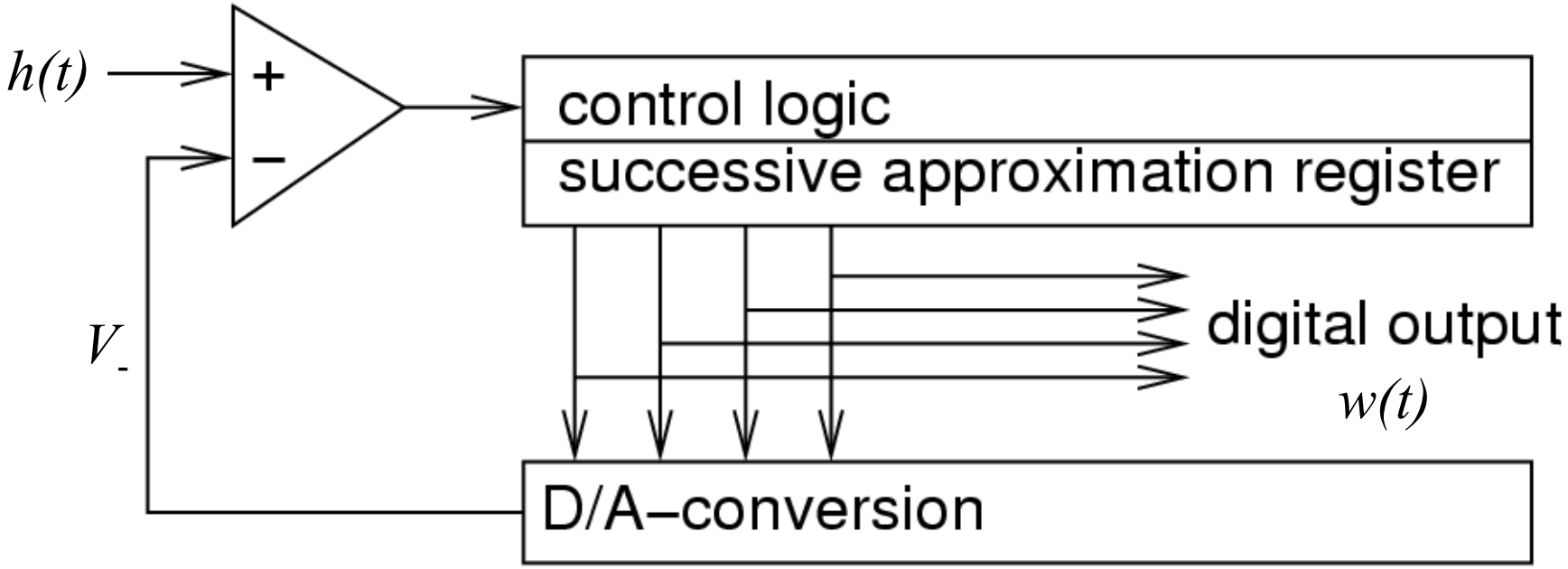
Parallel comparison with reference voltage

Speed: $O(1)$

Hardware complexity: $O(n)$

Applications: e.g. in video processing

Higher resolution: Successive approximation



Key idea: *binary search*

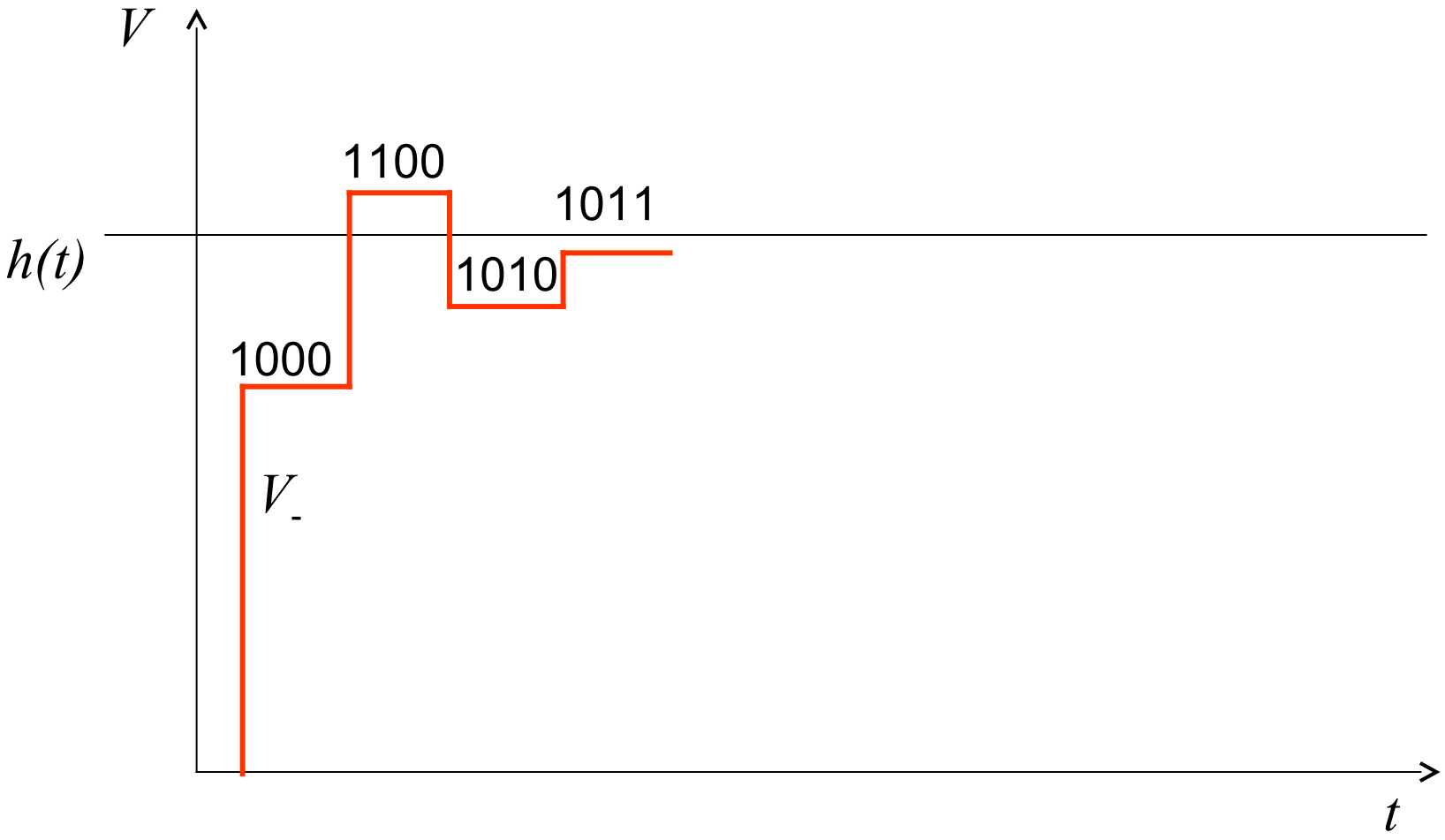
set MSB='1'
  if too large: reset MSB
set MSB-1='1'
  if too large: reset MSB-1
...

Speed: $O(\log_2(n))$

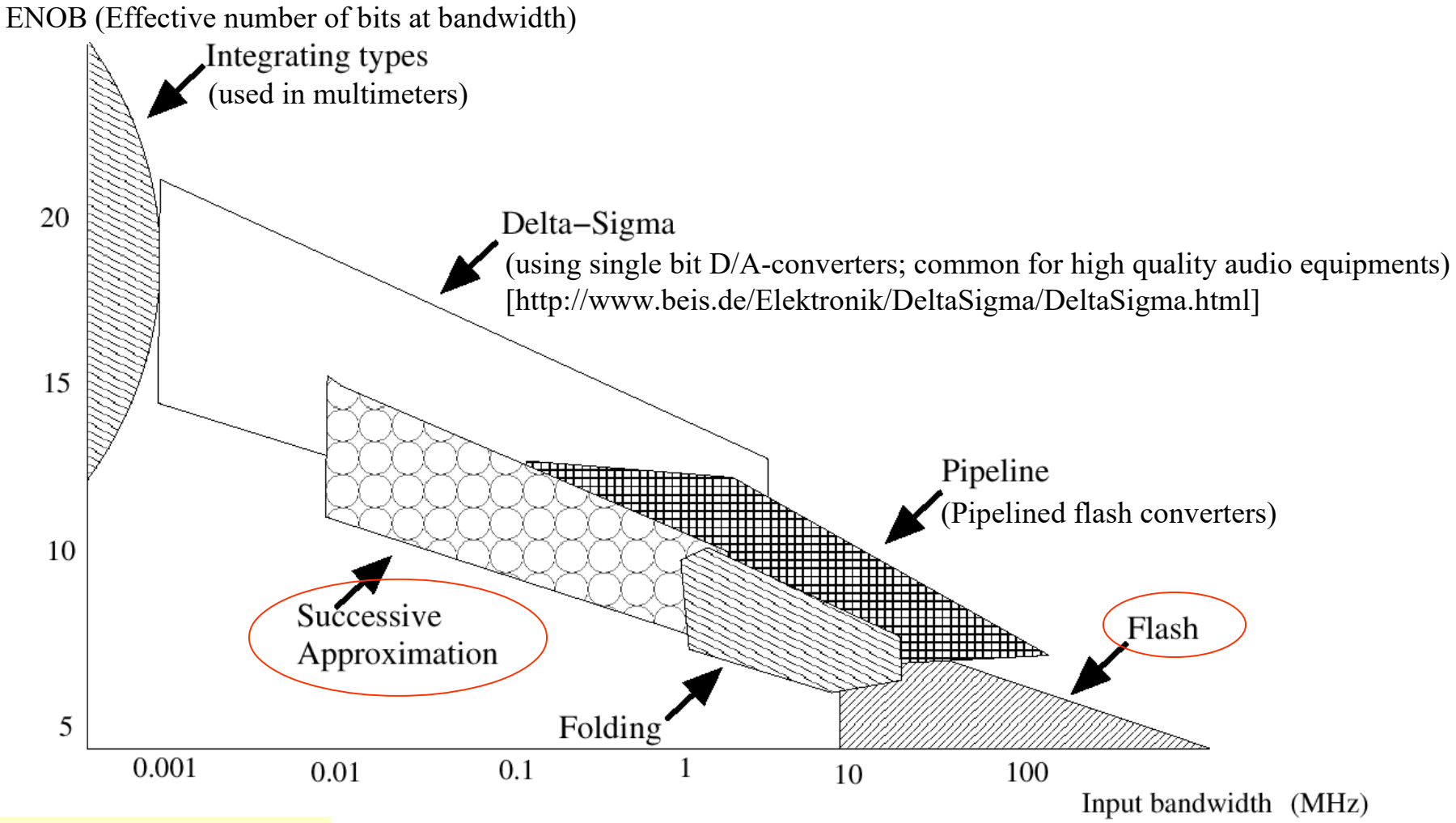
Hardware complexity: $O(\log_2(n))$

with n = # of voltage levels;
slow, but high precision possible.

Successive approximation (2)

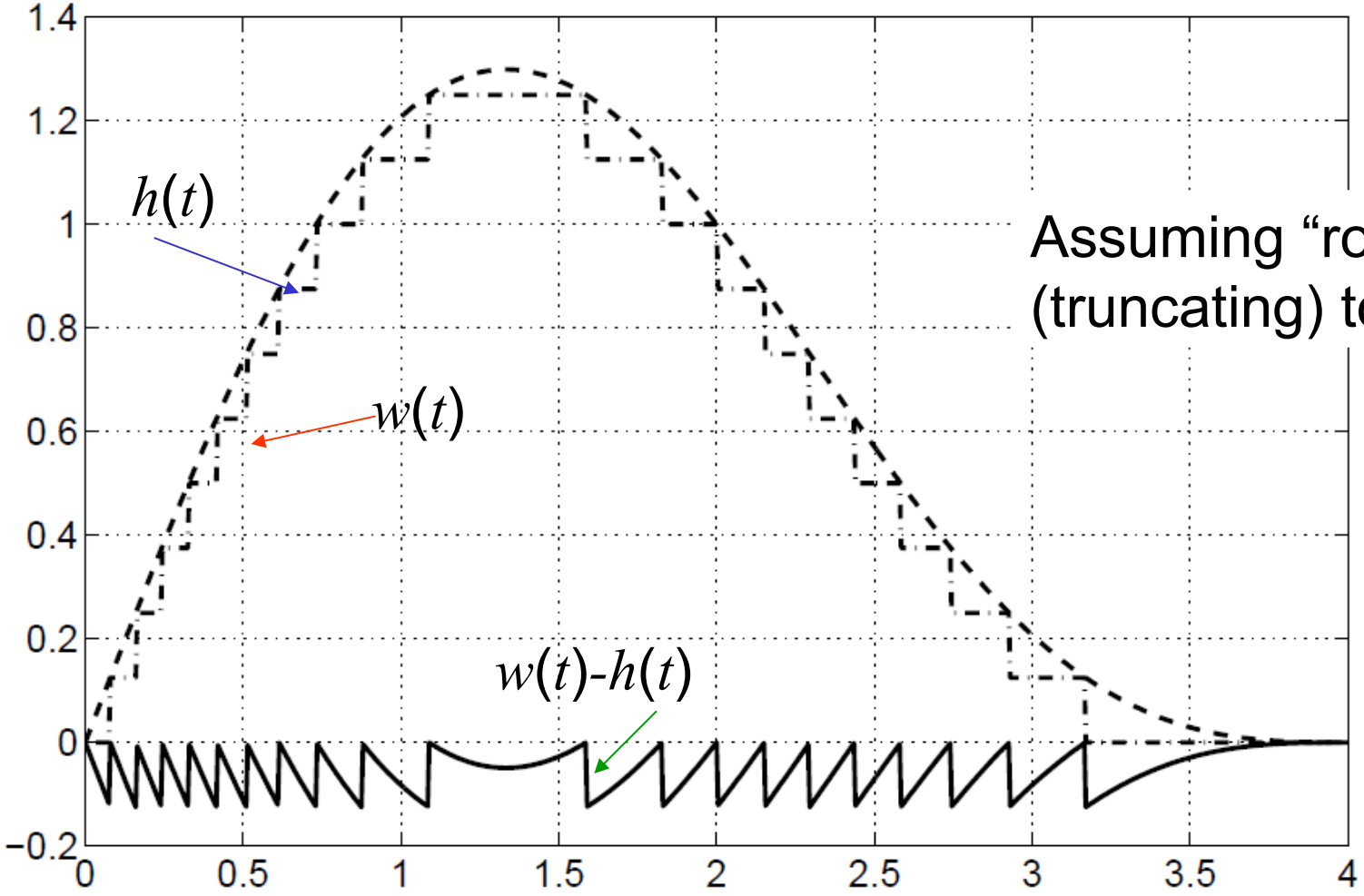


Application areas for flash and successive approximation converters



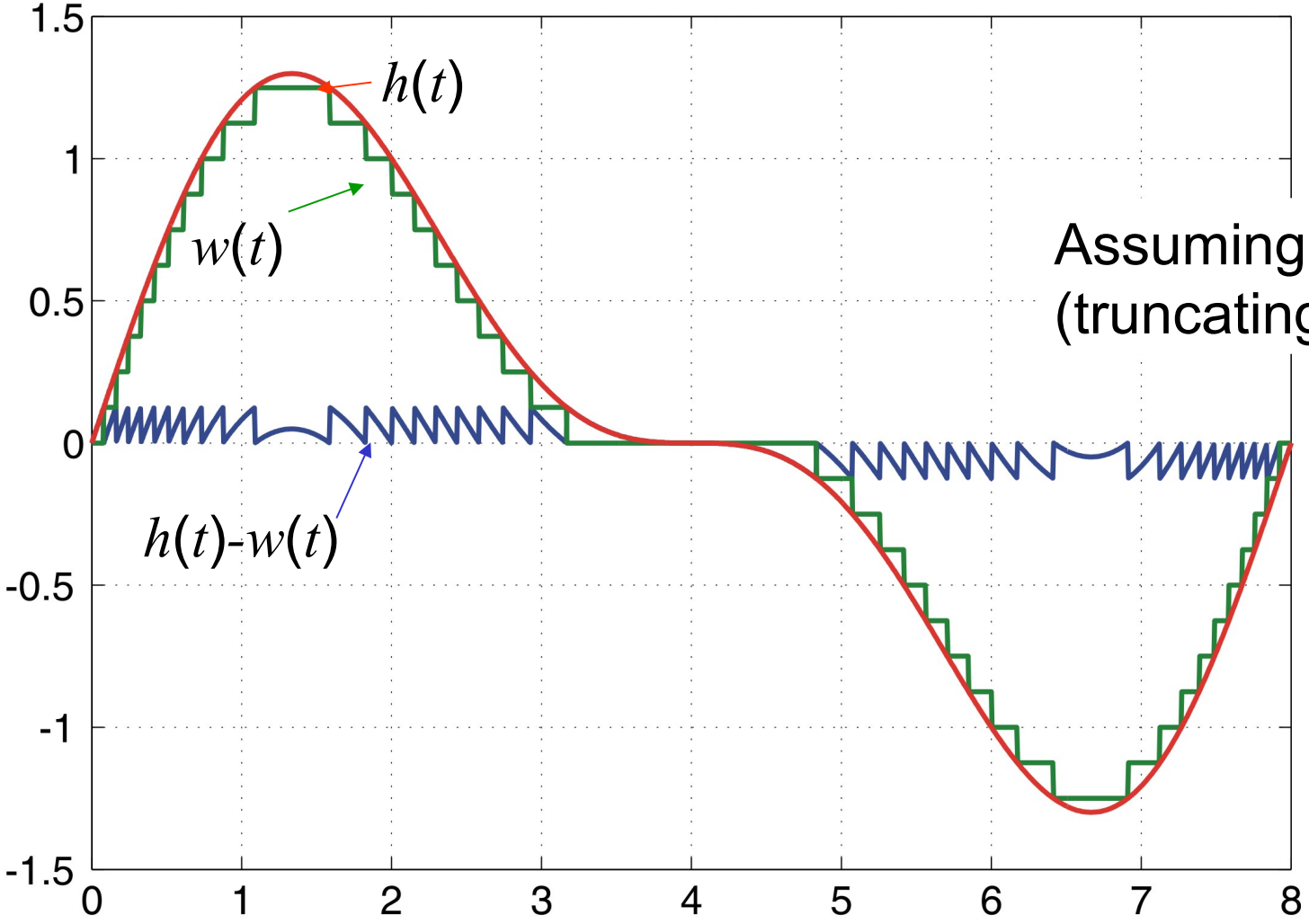
Gielen et al., DAC 2003

Quantization Noise



Assuming “rounding“ (truncating) towards 0

Quantization Noise



Assuming “rounding“ (truncating) towards 0

Signal to noise ratio

$$\text{Signal to noise ratio (SNR) [db]} = 20 \log_{10}\left(\frac{\text{effective signal voltage}}{\text{effective noise voltage}}\right)$$

e.g.: $20 \log_{10}(2) = 6.02$ decibels

Signal to noise for ideal n-bit converter : n * 6.02 + 1.76 [dB]
e.g. 98.1 db for 16-bit converter, ~ 160 db for 24-bit converter

Additional noise for non-ideal converters

Summary

Hardware in a loop

- Sensors
- Discretization
  - Sample-and-hold circuits
    - Aliasing (and how to avoid it)
    - Nyquist criterion
  - A/D-converters
    - Quantization noise

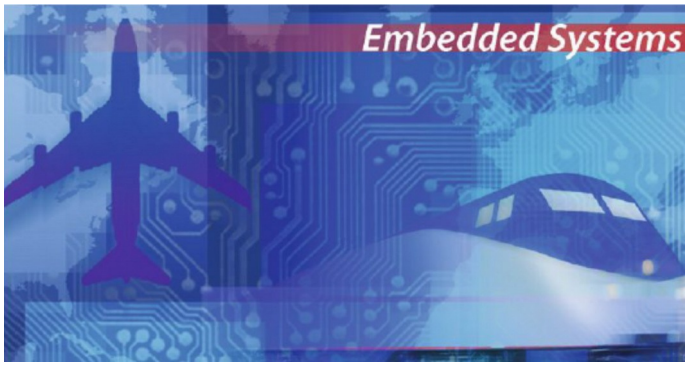
technische universität
dortmund

fakultät für informatik
informatik 12

Embedded System Hardware

- Processing -

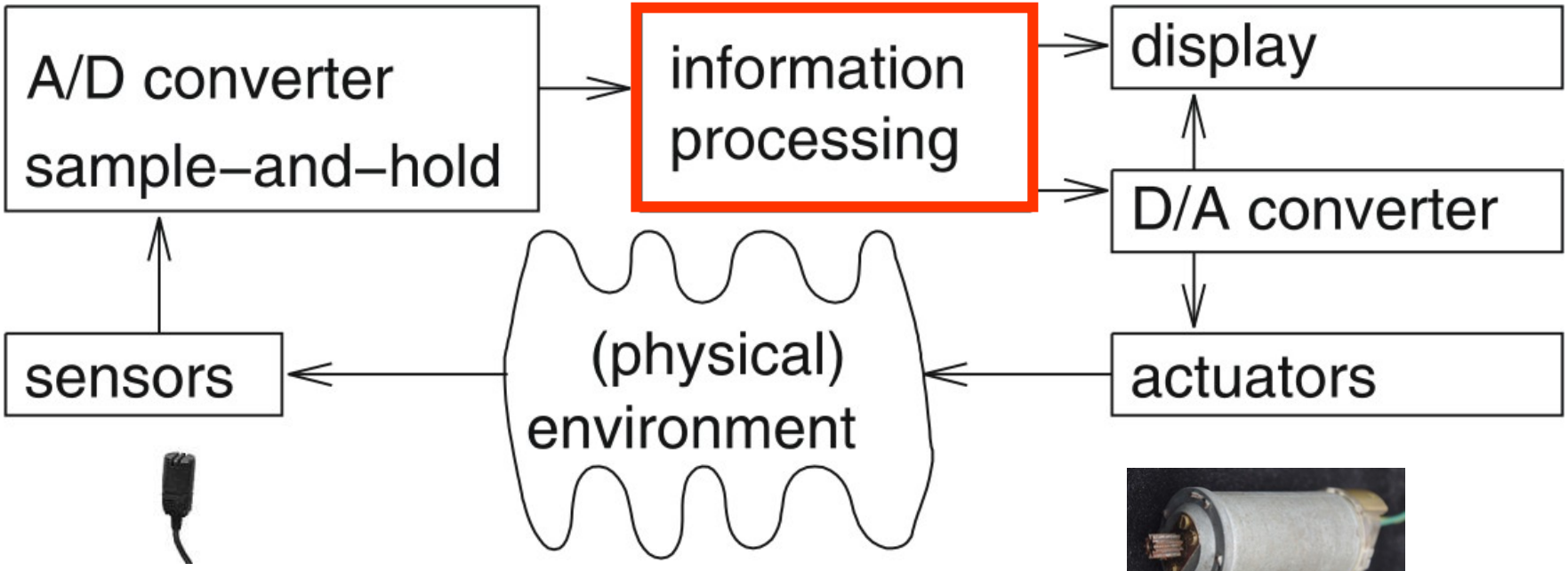
Peter Marwedel
TU Dortmund,
Informatik 12



2012 年 11 月 14 日

Embedded System Hardware

Embedded system hardware is frequently used in a loop (***"hardware in a loop"***):

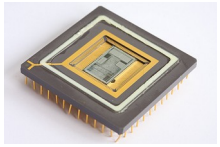


☞ cyber-physical systems

Efficiency:
slide from lecture 1 applied to processing

- CPS & ES must be **efficient**

- Code-size efficient
  (especially for systems on a chip)



- Run-time efficient

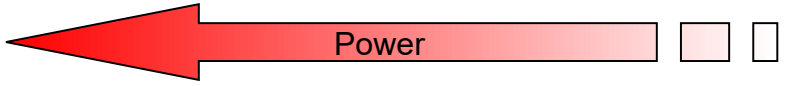
- Weight efficient

- Cost efficient

- Energy efficient

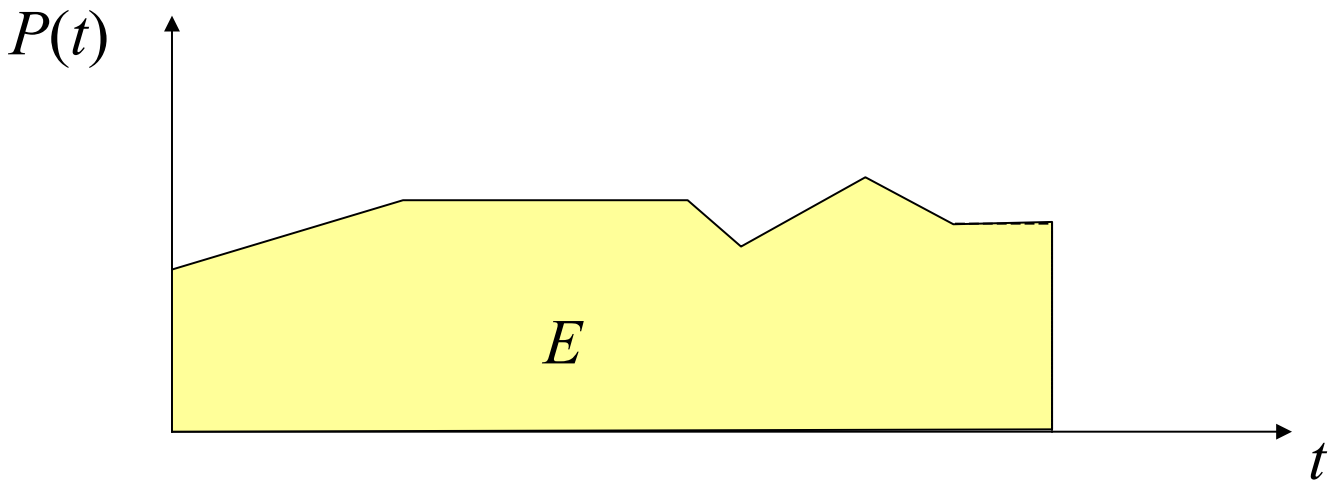
Key requirement: Energy and power efficiency - Why care about energy efficiency ?

	Relevant during use?		
Execution platform	Plugged	Uncharged periods	Unplugged
E.g.	Factory	Car	Sensor
Global warming	☑	☐	☐
Cost of energy	☑	☐	☐
Increasing performance	☑	☑	☑
Problems with cooling, avoiding hot spots	☑	☑	☑
Avoiding high currents & metal migration	☑	☑	☑
Reliability	☑	☑	☑
Energy a very scarce resource	☐	☑	☑



Should we care about energy consumption or about power consumption ? (1)

$$E = \int P(t)\, dt$$



Both are closely related, but still different

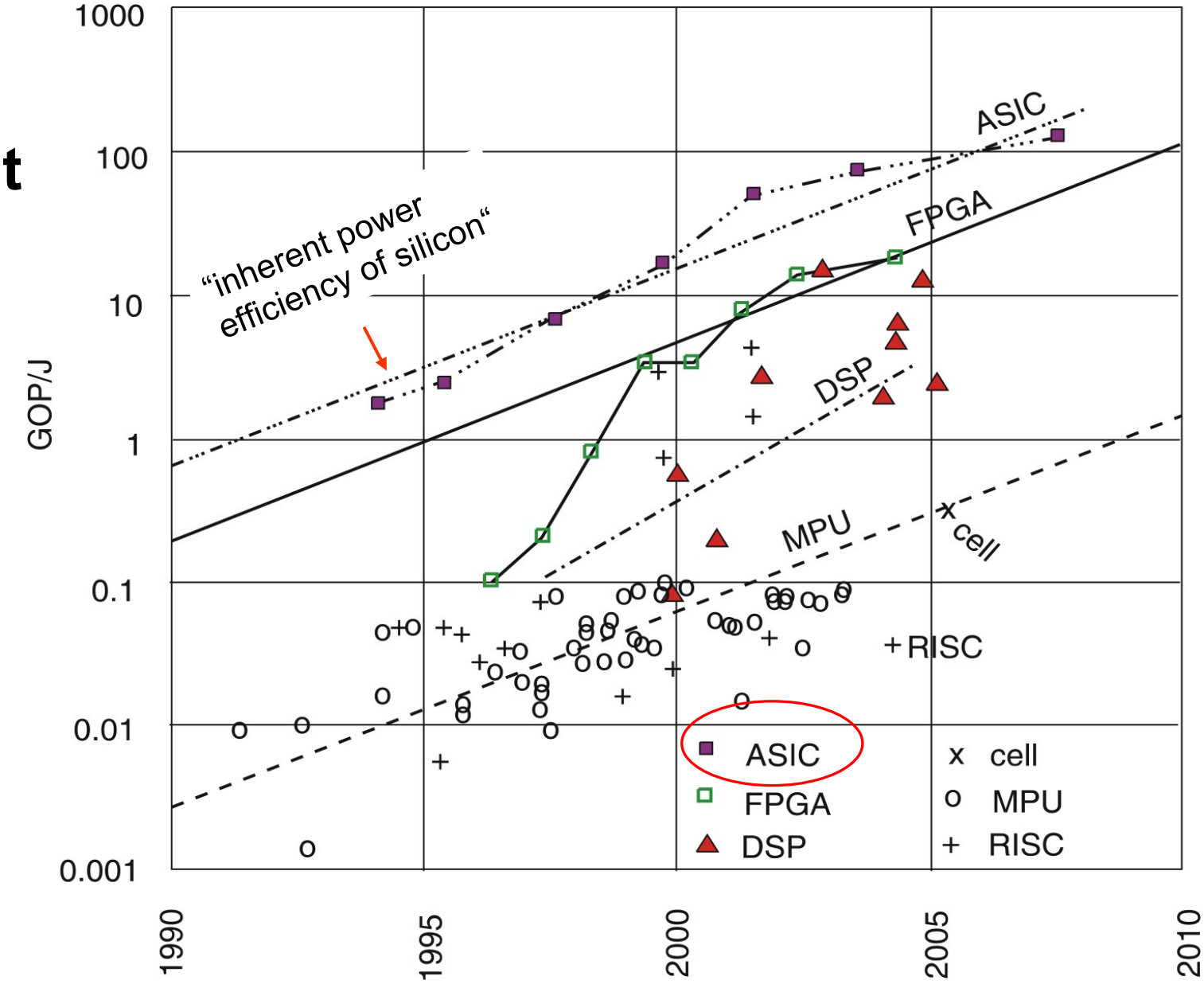
Should we care about energy consumption or about power consumption ? (2)

- Minimizing **power consumption** important for
  - design of the power supply & regulators
  - dimensioning of interconnect, short term cooling

- Minimizing **energy consumption** important due to
  - restricted availability of energy (mobile systems)
  - cooling: high costs, limited space
  - thermal effects
  - dependability, long lifetimes

☞ **In general, we need to care about both**

Energy Efficiency of different target platforms



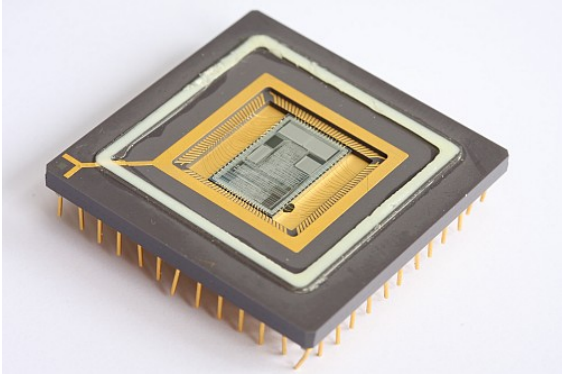
Application Specific Circuits (ASICS) or Full Custom Circuits

Approach suffers from

- long design times,
- lack of flexibility (changing standards) and
- high costs (e.g. Mill. $ mask costs).

Custom-designed circuits necessary

- if ultimate speed or
- energy efficiency is the goal and
- large numbers can be sold.



☞ HW synthesis not covered in this course, let's look at processors

Einschub: VLSI Entwurfsstile

– Beginn –

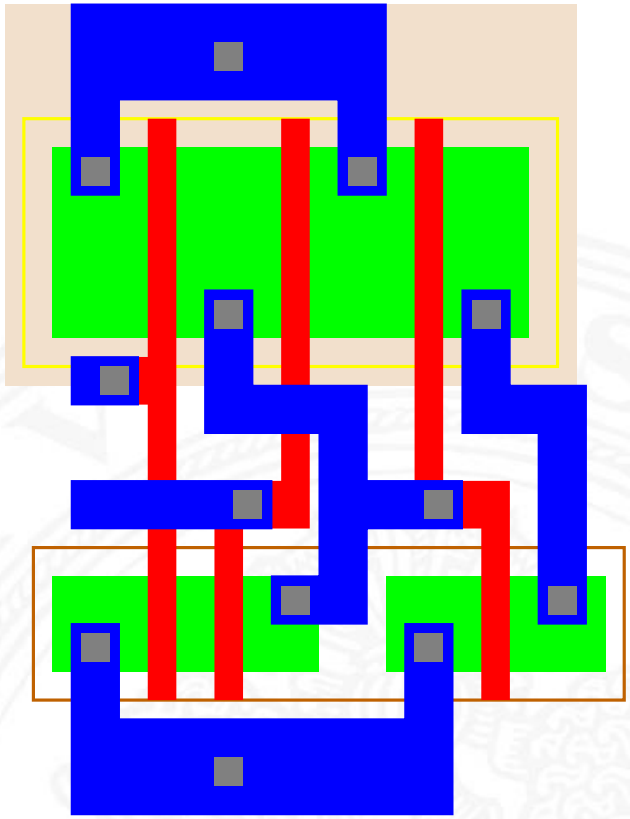
Entwurfsstile

- mehrere Möglichkeiten Schaltungen zu entwerfen
- Unterscheidungsmerkmale
  - Zeitaufwand: Entwurfsdauer, Fertigungszeit
  - Kosten: Fertigung, pro Stück, EDA-Werkzeuge
  - IC-Eigenschaften: Größe, Taktfrequenz, Leistungsaufnahme ...
- Entwurfsstile
  - Full-Custom
  - Standardzell
  - Gate-Array
  - FPGA / programmierbare Schaltungen

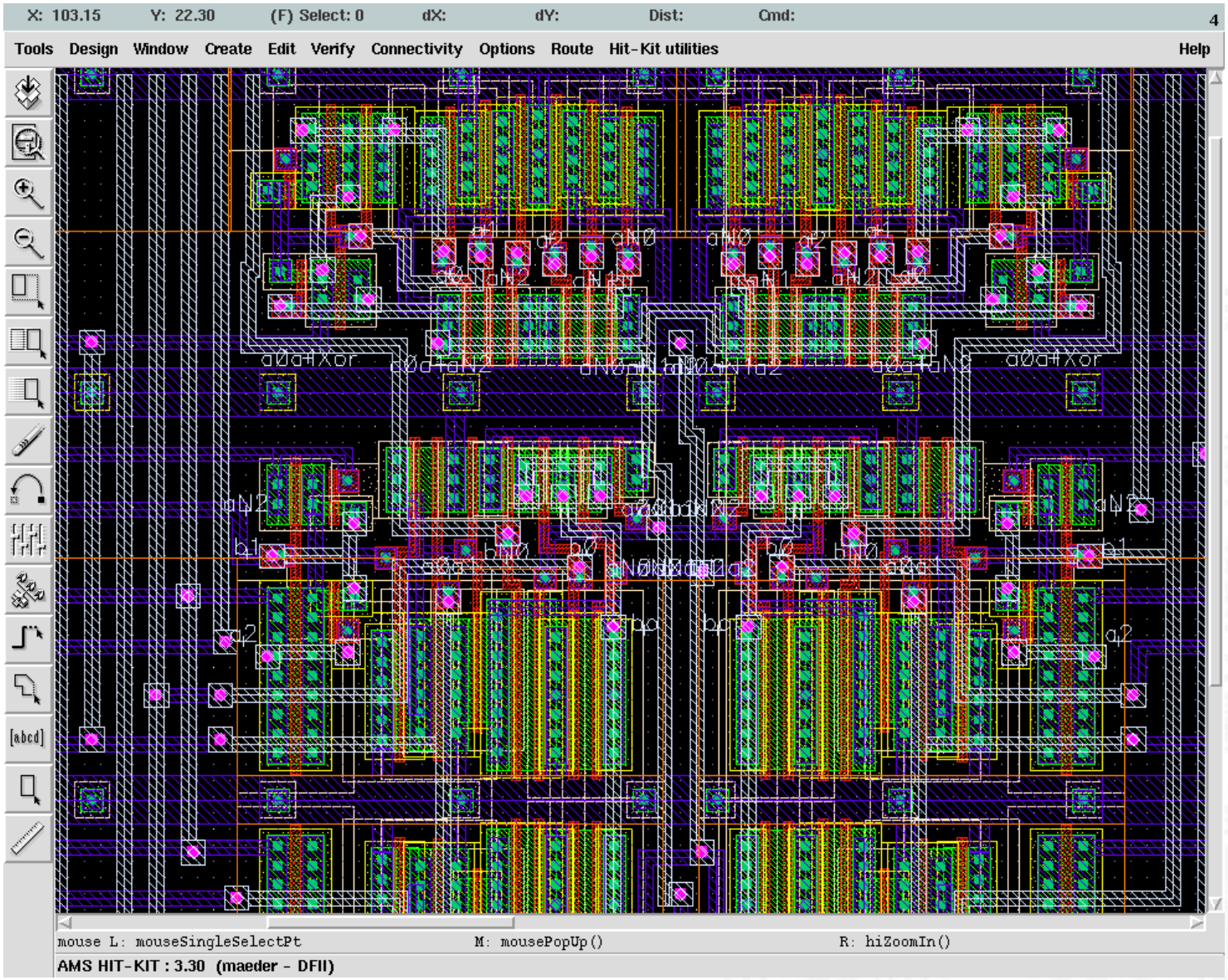
Full-Custom

Vollkundenspezifischer Entwurf / Full-Custom

- Layout aller geometrischer Strukturen
- viel manuelle Arbeit mit Layout-Editoren
- optimal kleine, schnelle Entwürfe
- sehr lange Entwurfsdauer (Effizienz)
- Ausnutzen von Regularität
- Teamarbeit nötig, Schnittstellen
- erfordert erfahrene Entwerfer



Full-Custom (cont.)

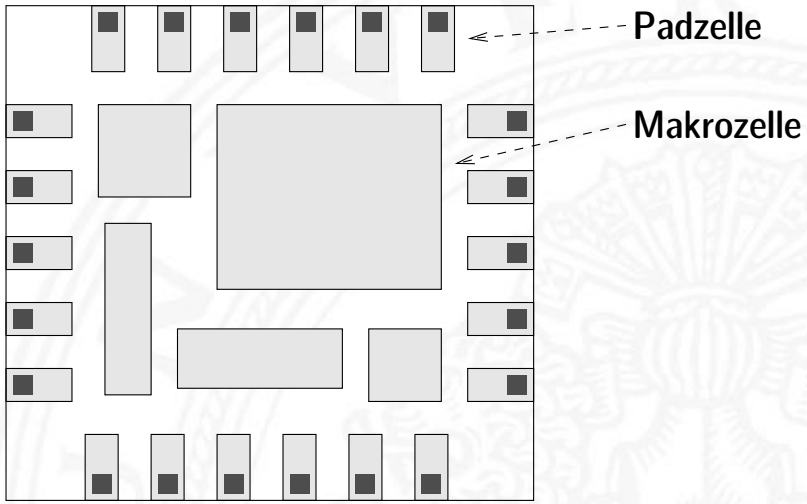


Makrozellentwurf

Makrozellentwurf

- Zellen wie Speicher, ALUs oder Datenpfade werden über Generatoren erzeugt
- Makrozellen in Full-Custom Qualität
- meist in Verbindung mit Standardzellentwurf

Chipgröße	variabel
Zellenanzahl	variabel
Zellengröße	variabel
Anschlusslage	variabel
Leiterbahnkanäle	variabel

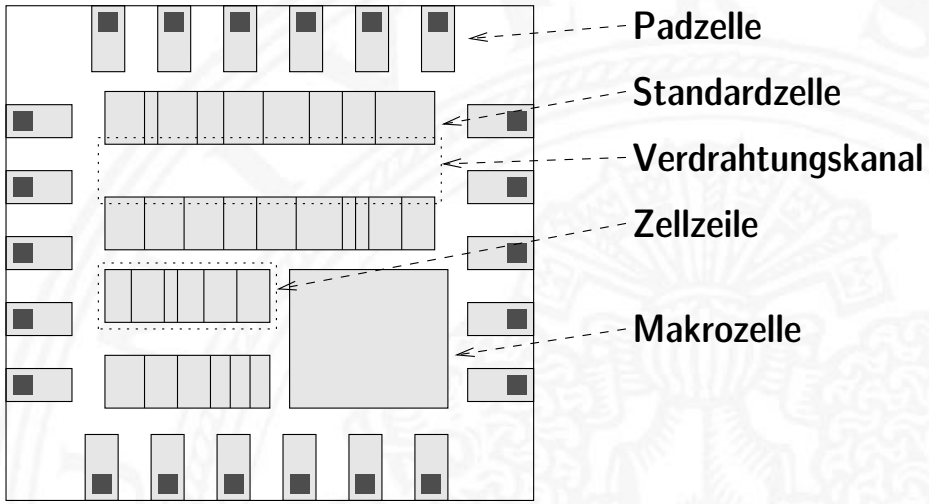


Standardzellentwurf

Standardzellentwurf

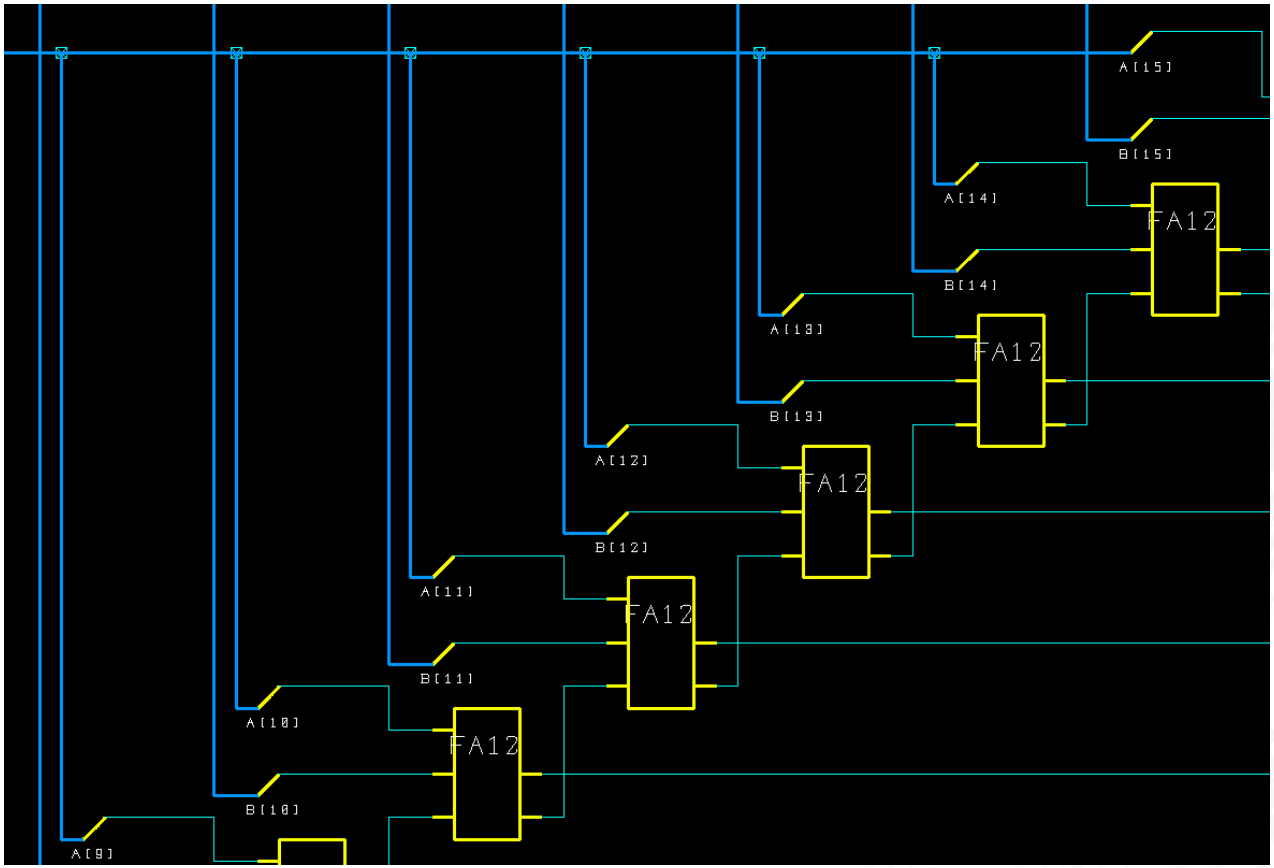
- vorgefertigte Zellen aus Bibliotheken benutzen
- Layout der Standardzellen in Full-Custom Qualität
- schneller flexibler Entwurf
- meist in Verbindung mit Makrozellgeneratoren

Chipgröße	variabel
Zellenanzahl	variabel
Zellenhöhe	fest
Zellenbreite	variabel
Anschlusslage	variabel
Leiterbahnkanäle	variabel

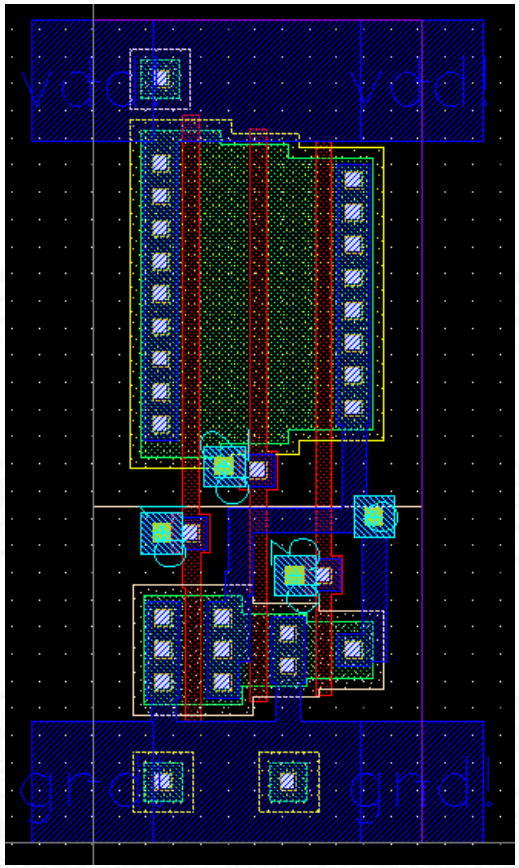


Standardzellentwurf (cont.)

Schematic

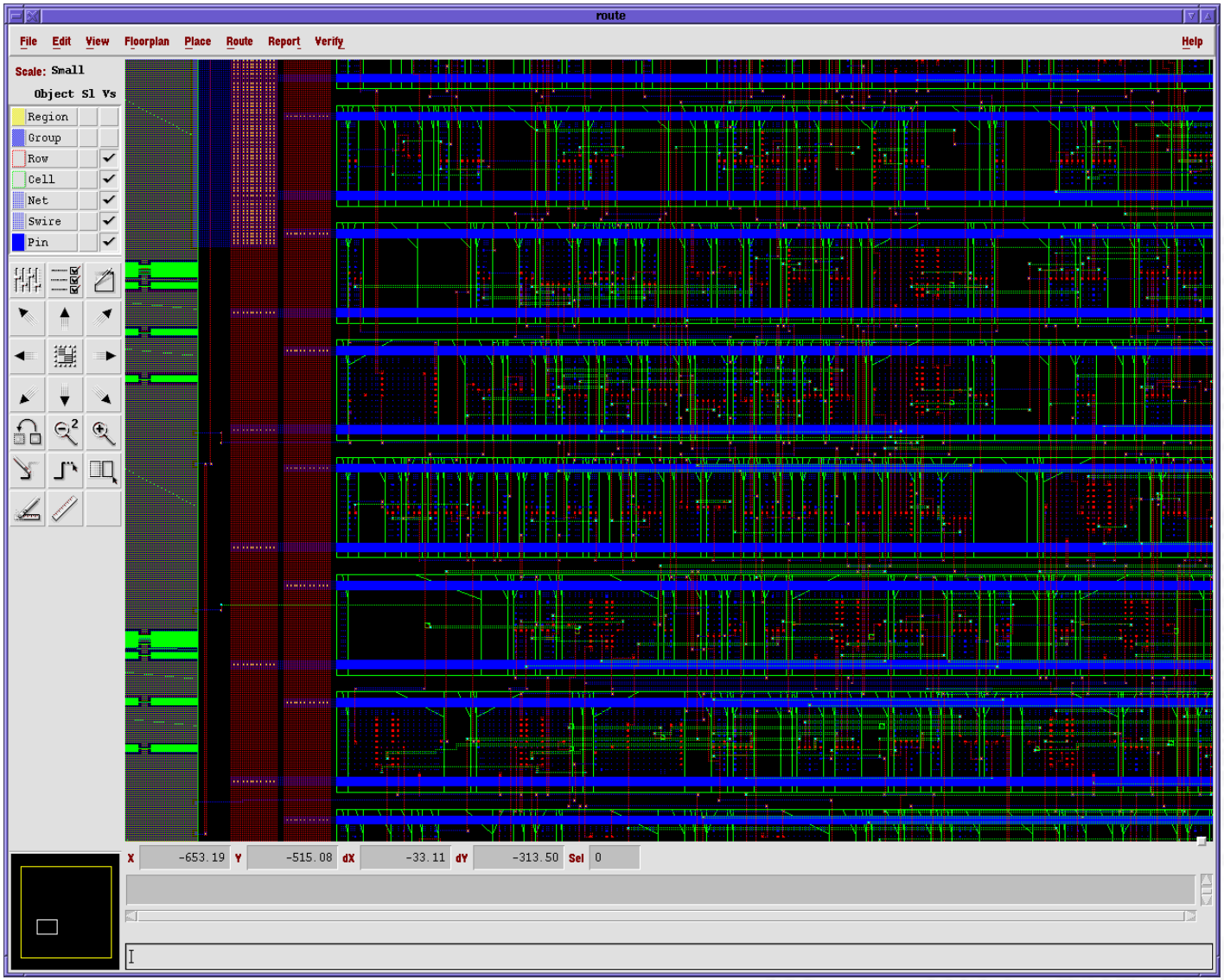


Zell-Layout



Standardzellentwurf (cont.)

Standardzell Layout

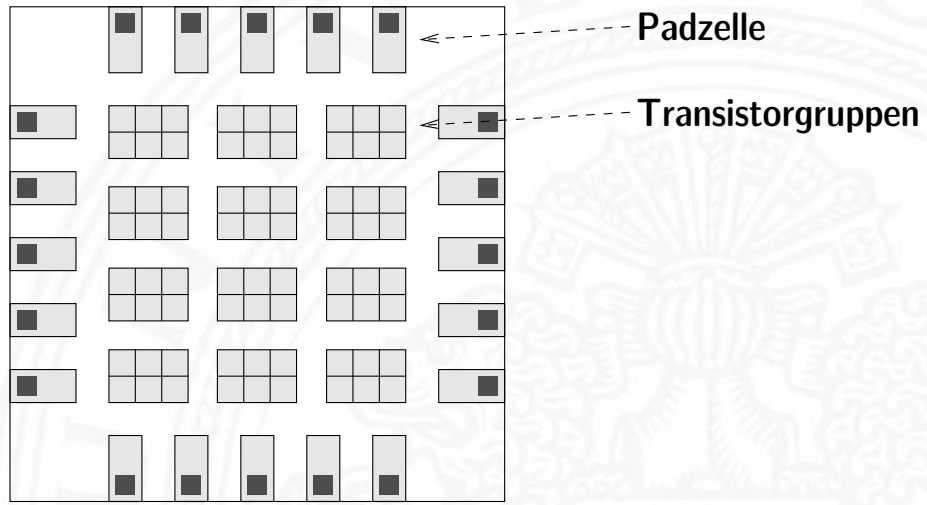


Gate-Array Entwurf

Gate-Array / Sea-of-Gate Entwurf — abgelöst durch FPGA

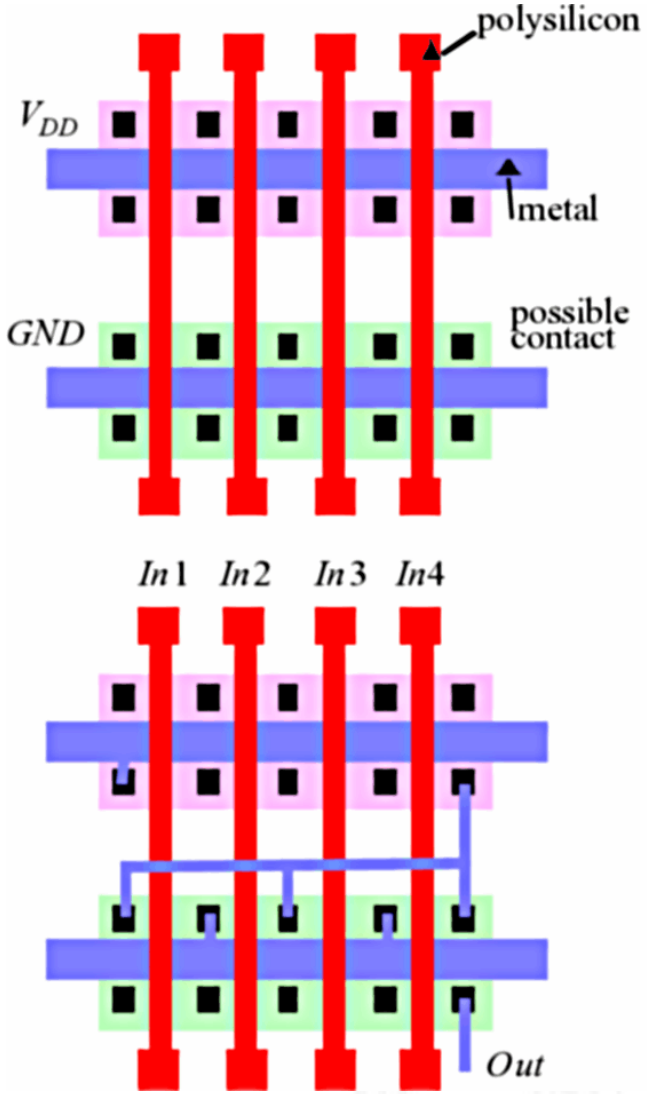
- vorgefertigte Transistoren
- Layout durch Verbindungsstruktur (Verdrahtung, Kontakte)
- intra-Zell Verdrahtung aus Zellbibliotheken
- vorgegebene Master: Komplexität eingeschränkt, Verschnitt
- schnelle Verfügbarkeit

Chipgröße	fest
Zellenanzahl	fest
Zellengröße	fest
Anschlusslage	fest
Leiterbahnkanäle	fest



Gate-Array Entwurf (cont.)

Gate-Array



Uncommited Cell

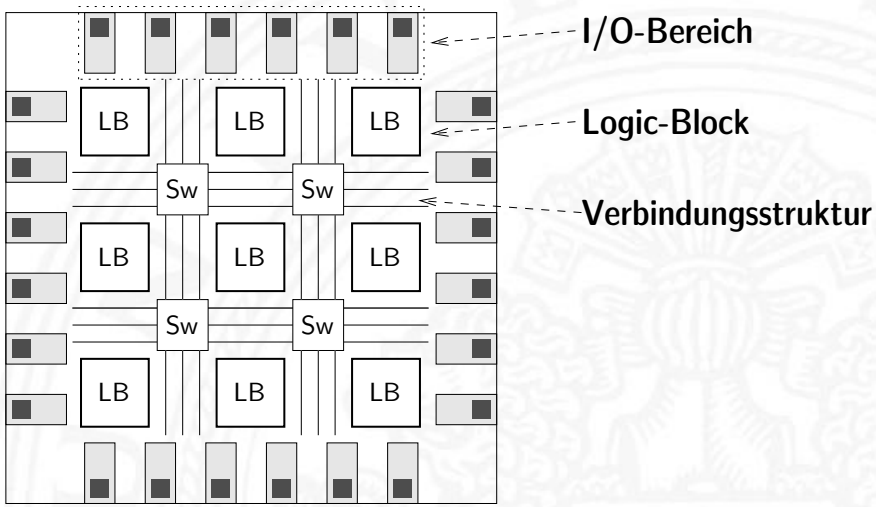
Committed Cell (4-input NOR)

programmierbare Schaltungen

programmierbare Schaltungen: FPGA, PLD, LCA ...

- fertig vorgegebene Schaltung: Logik und Verbindungsstruktur
- Entwurf: Programmierung durch Anwender $\Rightarrow$ sofort verfügbar
- Einschränkung durch vorgegebene Struktur
- Rekonfiguration möglich
- in-Circuit programmierbar

Chipgröße	fest
Blockanzahl	fest
Anschlusslage	fest
Verbindungsnetz	fest
Blockfunktion	progr.
Verbindungen	progr.



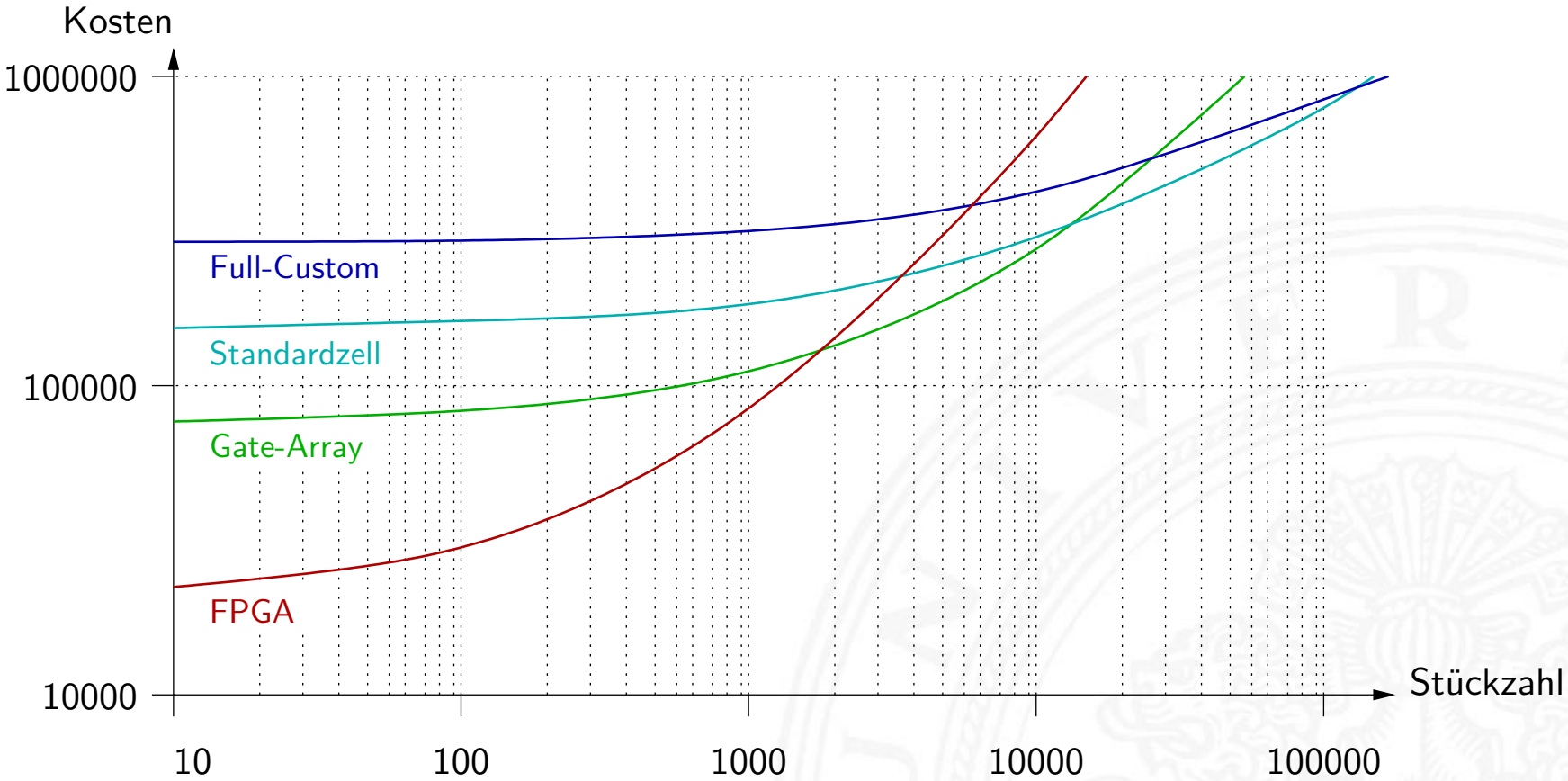
Vergleich der Entwurfsstile

Tabellarische Übersicht

Stil	*Performanz*	*Fläche*	*Kosten (IC)*	*Kosten (Design)*	*time-to-Market*	*Prozessschritte*	*Stückzahlen*
Full-Custom	$+++$	$+++$	$+++$	$---$	$---$	voll	10^5
Standard-/Makrozell	$++$	$++$	$++$	$--$	$--$	voll	10^4
Gate-Array	$+$	$\circ$	$+$	$\circ$	$\circ$	4-10	10^3
programmierbare Logik	$-$	$--$	$--$	$++$	$+++$	0	$< 10^3$

Vergleich der Entwurfsstile (cont.)

Wirtschaftlichkeitsüberlegungen – Tendenz !



Vergleich der Entwurfsstile (cont.)

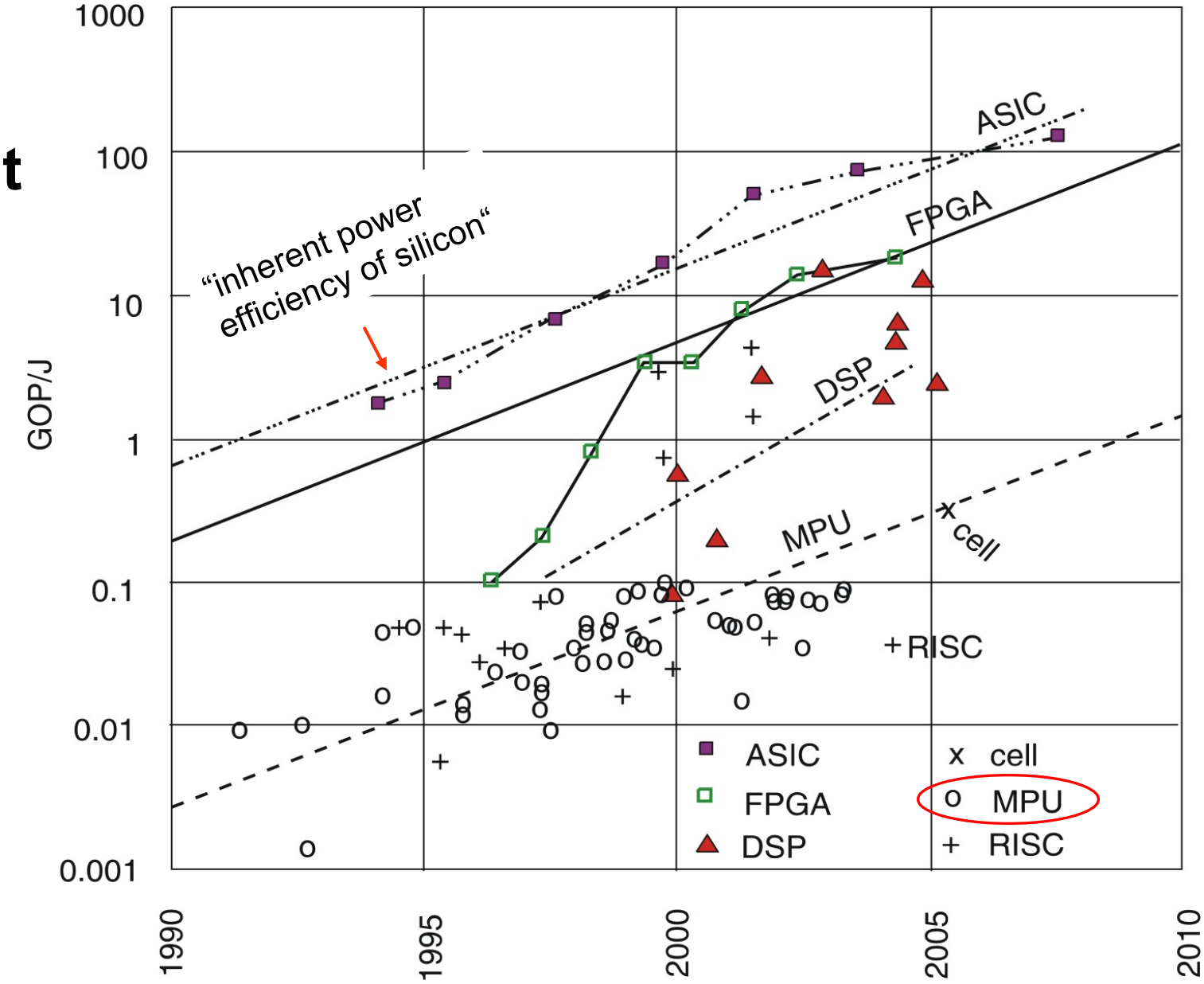
Faktoren bei der Auswahl

- Kostenüberlegungen
- Entwurfsdauer: „time-to-Market“
- technische Randbedingungen, oft als K.O.-Kriterium
  - Fläche
  - Leistungsaufnahme
  - Sicherheitsaspekte
- organisatorische Randbedingungen
  - vorhandene Werkzeuge
  - Know-How
  - „Faktor: Mensch“ (Erfahrungen, Vorlieben)

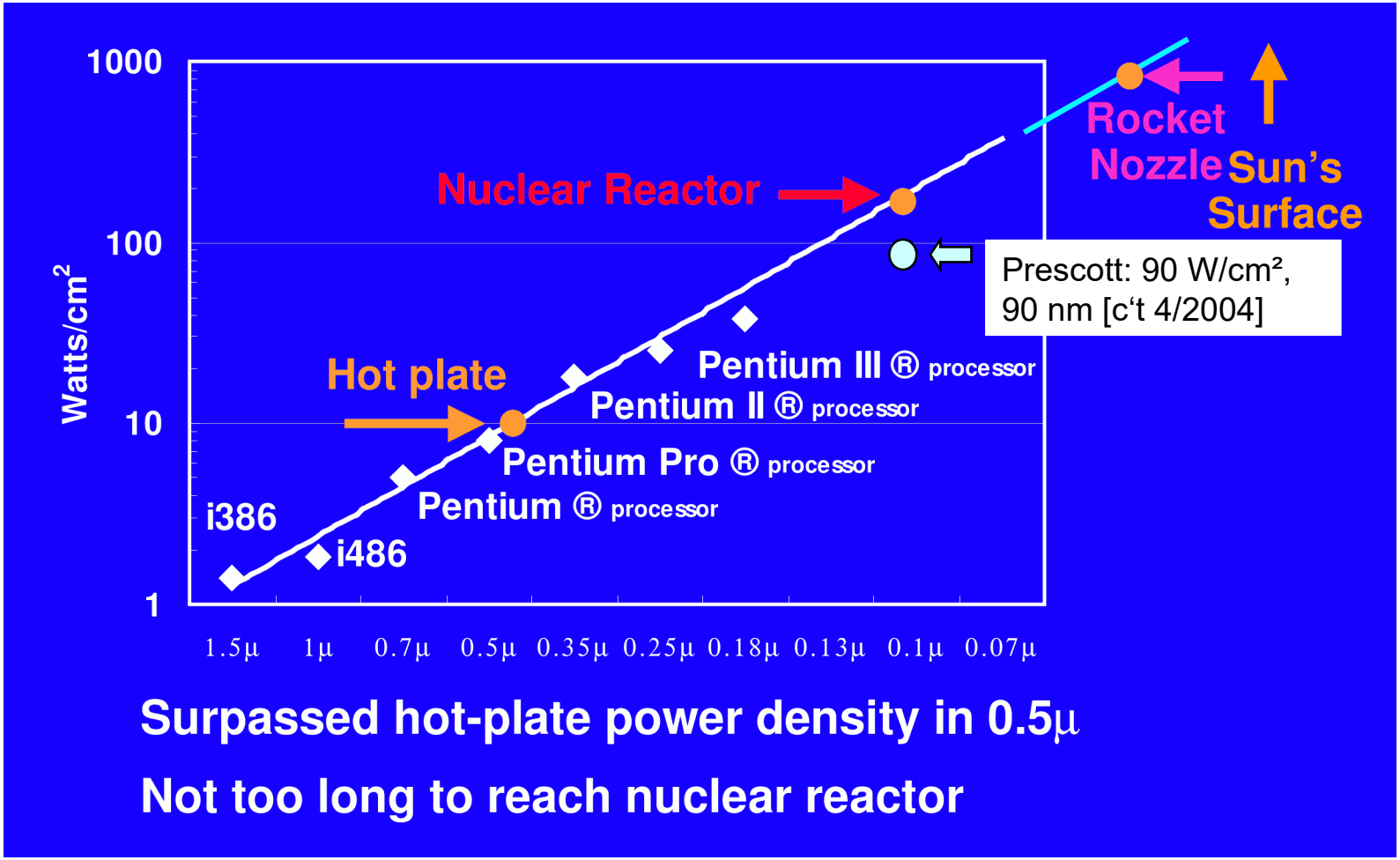
$\Rightarrow$ vielfältige Wechselwirkungen

– Ende –

Energy Efficiency of different target platforms

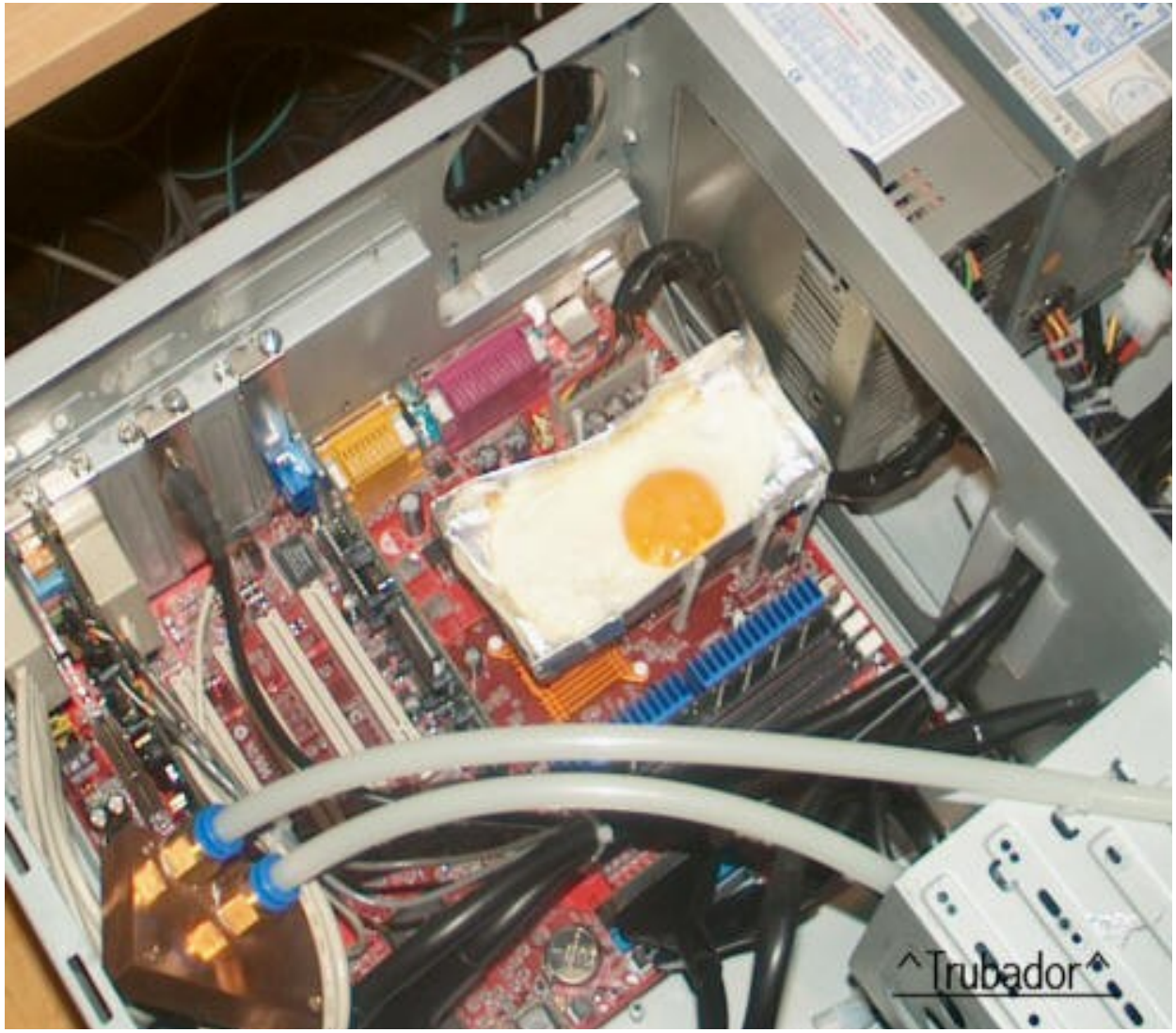


Problem:
Power density continues to get worse



- aktuell: 50...75 W/cm² [A.Mäder]

Surpassed hot (kitchen) plate …? Why not use it?

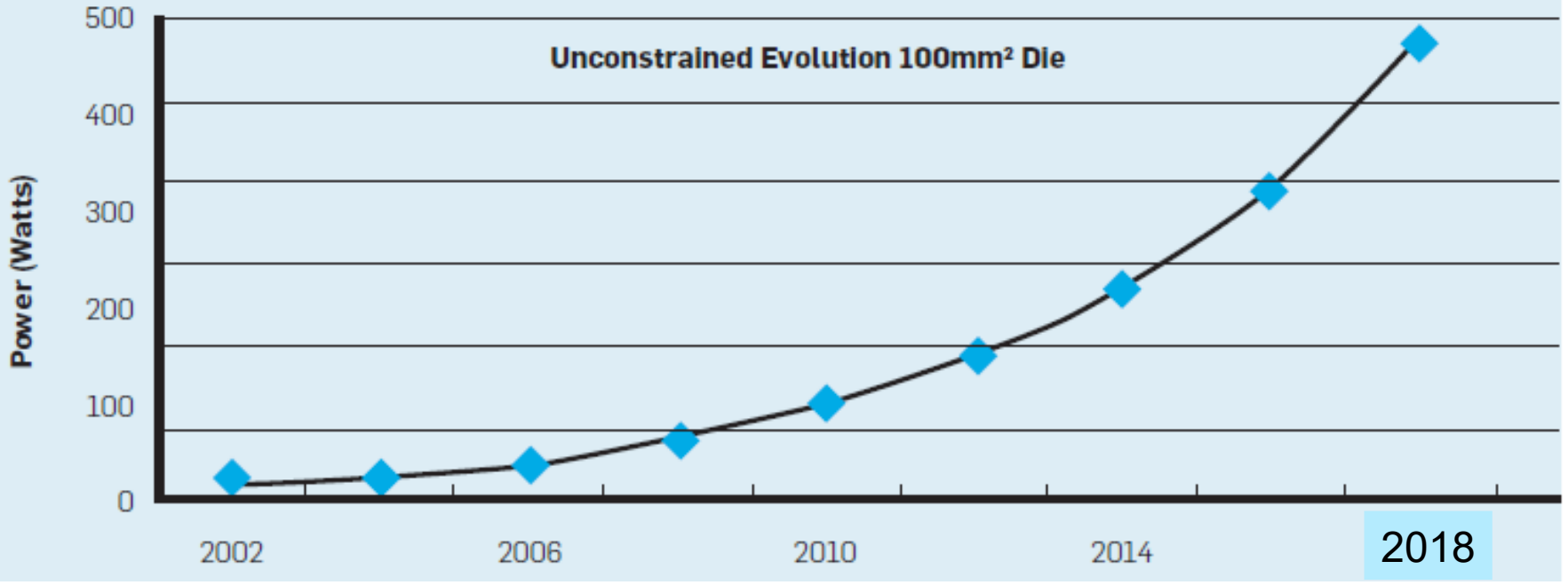


Strictly speaking, energy is not "consumed", but converted from electrical energy into heat energy

http://www.phys.ncku.edu.tw/~htsu/humor/fry_egg.html

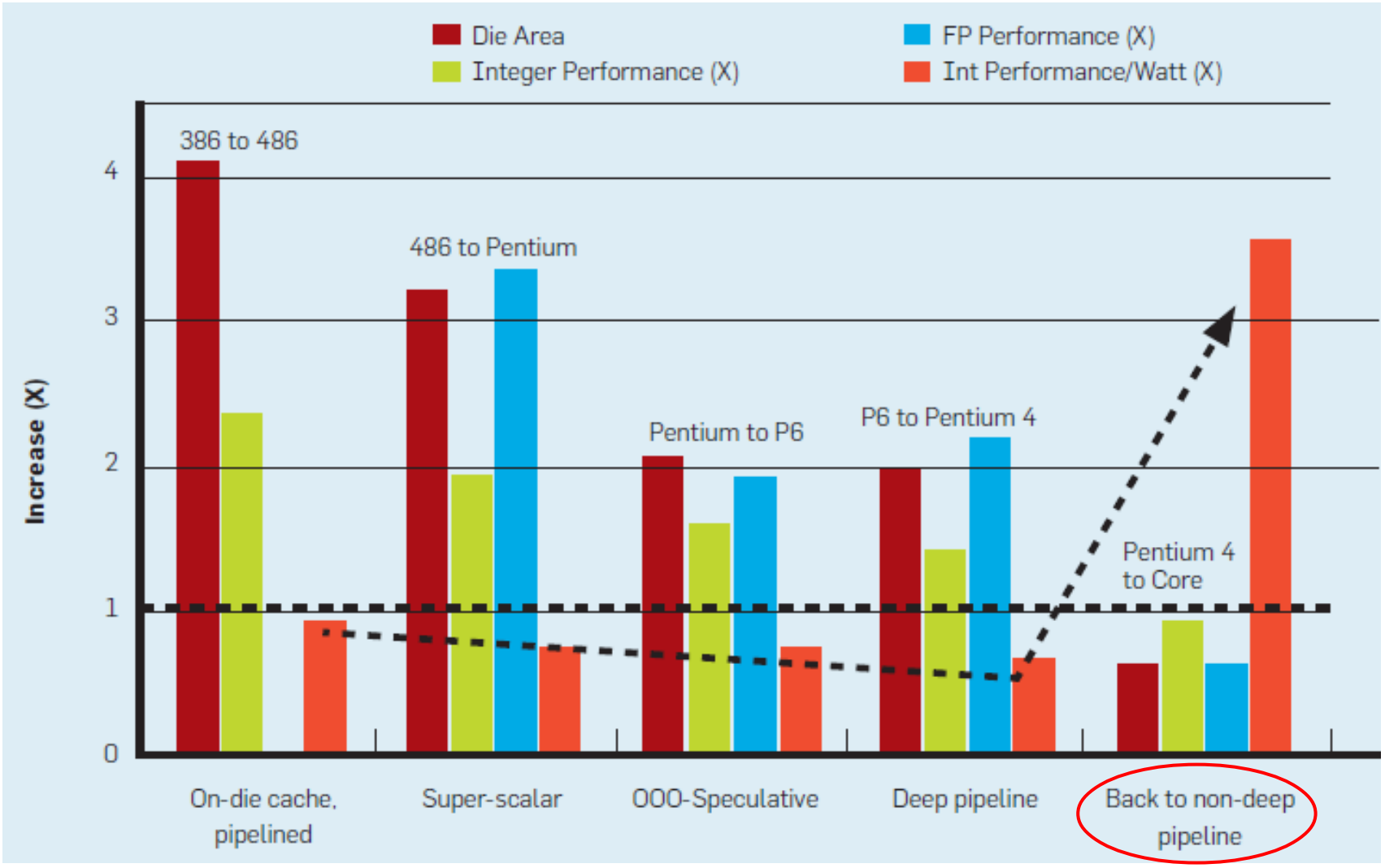
PCs: Just adding transistors would have resulted in this

Figure 7. Unconstrained evolution of a microprocessor results in excessive power consumption.



S. Borkar, A. Chien: The future of microprocessors, *Communications of the ACM*, May 2011

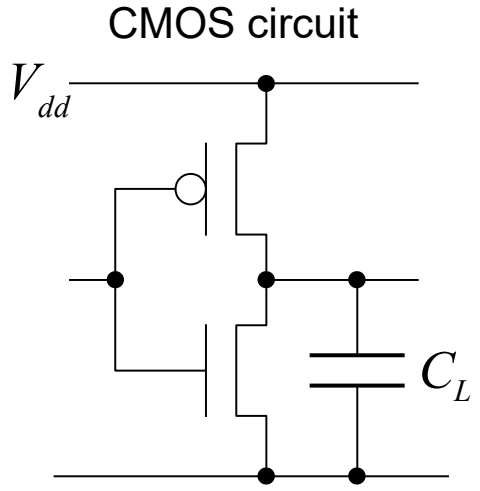
Keep it simple, stupid (KISS)



S. Borkar, A. Chien: The future of microprocessors, *Communications of the ACM*, May 2011

Static and dynamic power consumption

- Dynamic power consumption: Power consumption caused by charging capacitors when logic levels are switched.



$P = \alpha\, C_L\, V_{dd}^{\,2}\, f$ with

α : switching activity

C_L : load capacitance

V_{dd} : supply voltage

f : clock frequency

☞ Decreasing V_{dd} reduces P quadratically

- Static power consumption (caused by leakage current): power consumed in the absence of clock signals
- Leakage becoming more important due to smaller devices

Einschub: CMOS Schaltungen

– Beginn –

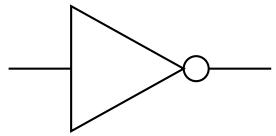
CMOS: Schaltungsprinzip von „static CMOS“

- Transistoren werden als Schalter betrachtet
- zwei zueinander **komplementäre** Zweige der Schaltung
  - n-Kanal Transistoren zwischen Masse und Ausgang y *1 on*
  - p-Kanal –"– Vdd und Ausgang y *0 on*

  - p-Kanal Zweig komplementär („dualer Graph“) zu n-Kanal Zweig: jede Reihenschaltung von Elementen wird durch eine Parallelschaltung ersetzt (und umgekehrt)

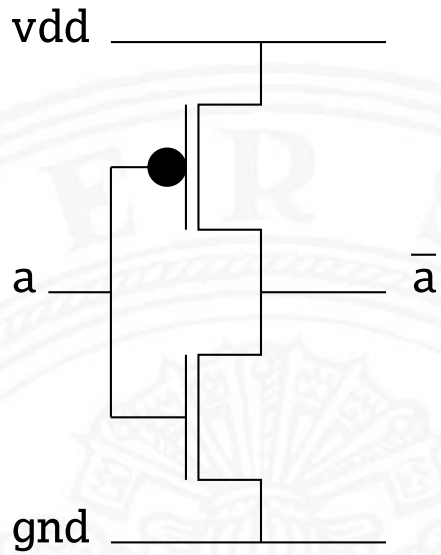
- immer ein direkt leitender Pfad von entweder Vdd („1“) oder Masse / Gnd („0“) zum Ausgang
- niemals ein direkt leitender Pfad von Vdd nach Masse
- kein statischer Stromverbrauch im Gatter

Inverter



Funktionsweise

- selbstsperrende p- und n-Kanal Transistoren
- komplementär beschaltet
- Ausgang: Pfad über p-Transistoren zu Vdd
  –"– n-Transistoren zu Gnd
- genau *einer* der Pfade leitet



- Eingang Trans$_P$ Trans$_N$ Ausgang

$a = 0 \rightarrow$ leitet / sperrt $\rightarrow$ über T$_P$ mit Vdd verbunden $= 1$

$a = 1 \rightarrow$ sperrt / leitet $\rightarrow$ über T$_N$ mit Gnd verbunden $= 0$

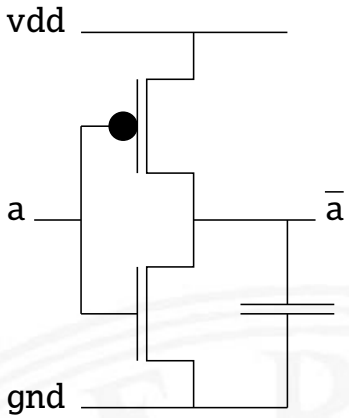
Inverter (cont.)

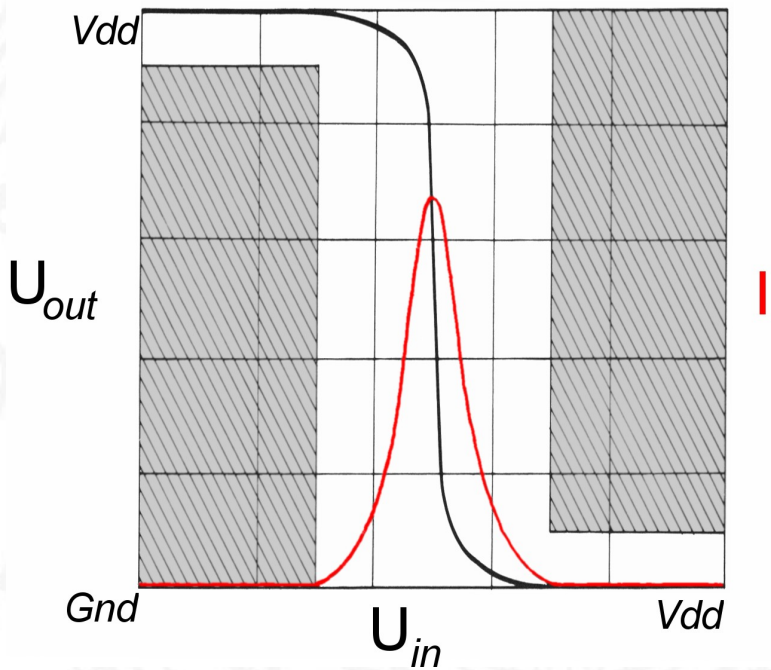
Leistungsaufnahme

1. $U_{in} = 0$, bzw. Vdd: Sperrstrom, nur μA
   $\Rightarrow$ niedrige statische Leistungsaufnahme
2. Querstrom beim Umschalten:
   kurzfristig leiten beide Transistoren
   $\Rightarrow$ Forderung nach steilen Flanken
3. Kapazitive Last: Fanout-Gates

   Energie auf Gate(s): $W = \frac{1}{2} C_T Vdd^2$

   Verlustleistung$_{(0/1/0)}$: $P = C_T Vdd^2 \cdot f$

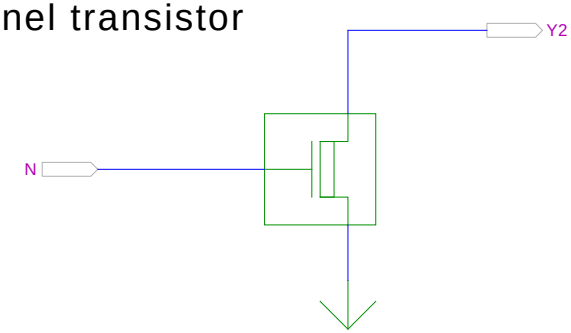




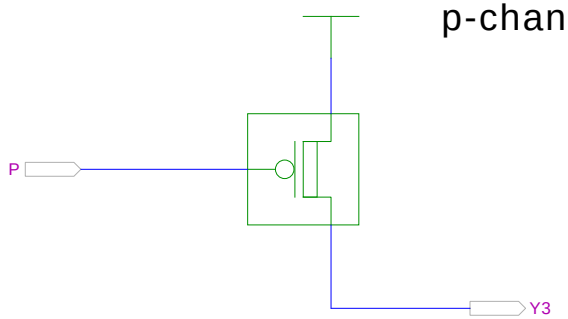
$\Rightarrow$ Transfercharakteristik

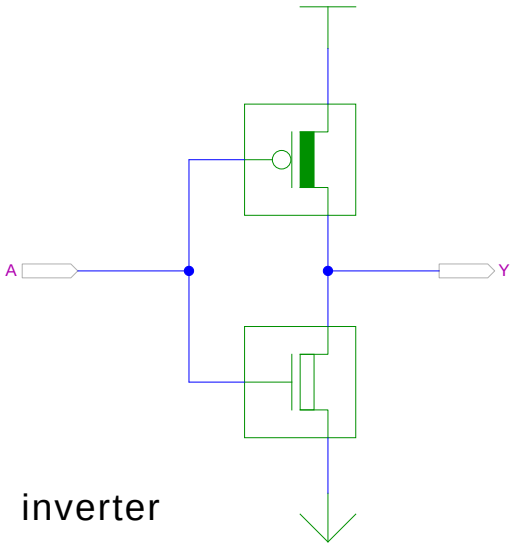
Hades: n- und p-Kanal Transistor, Inverter, Verstärker

n-channel transistor



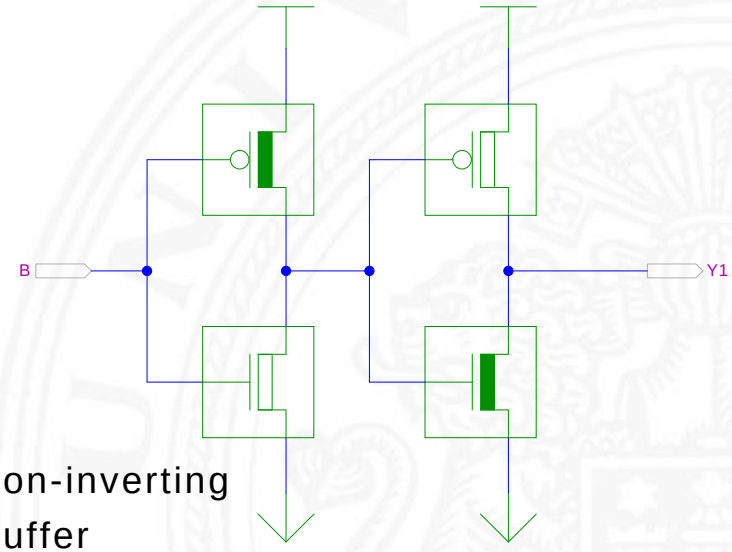
p-channel transistor





inverter

$$Y = \overline{A}$$

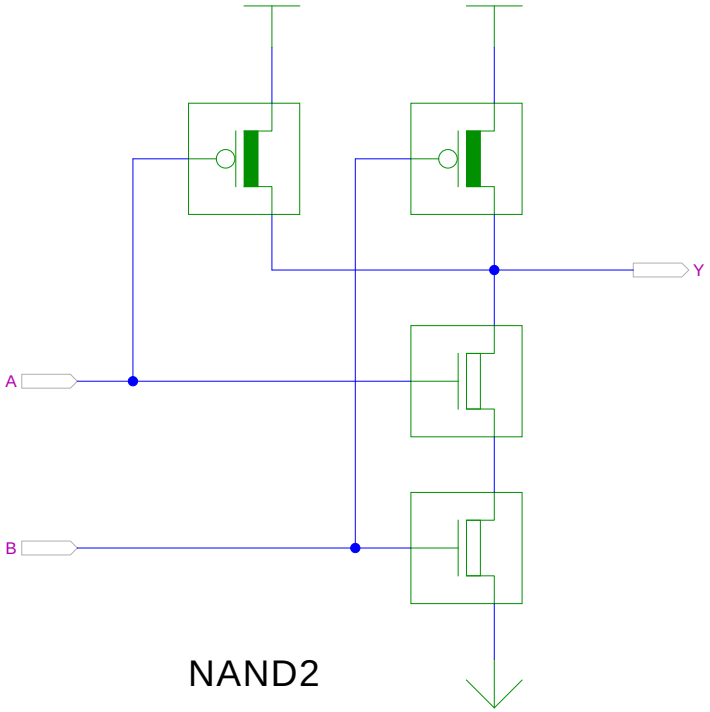


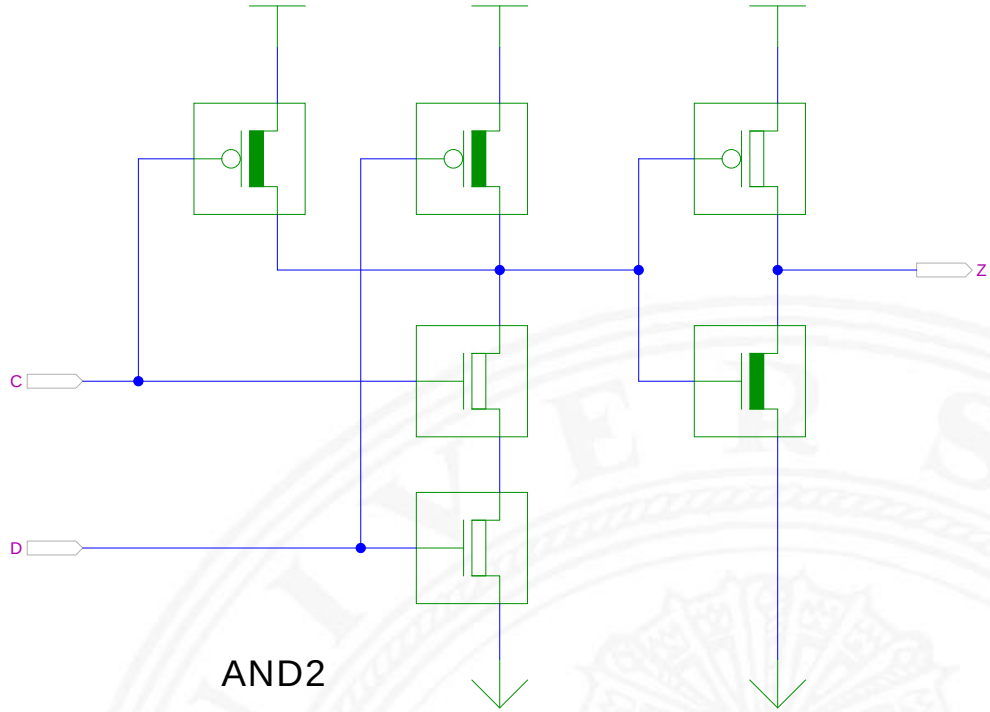
non-inverting buffer

$$Y = \overline{(\overline{A})} = A$$

[HenHA] Hades Demo: `05-switched/40-cmos/inverter`

NAND- und AND-Gatter

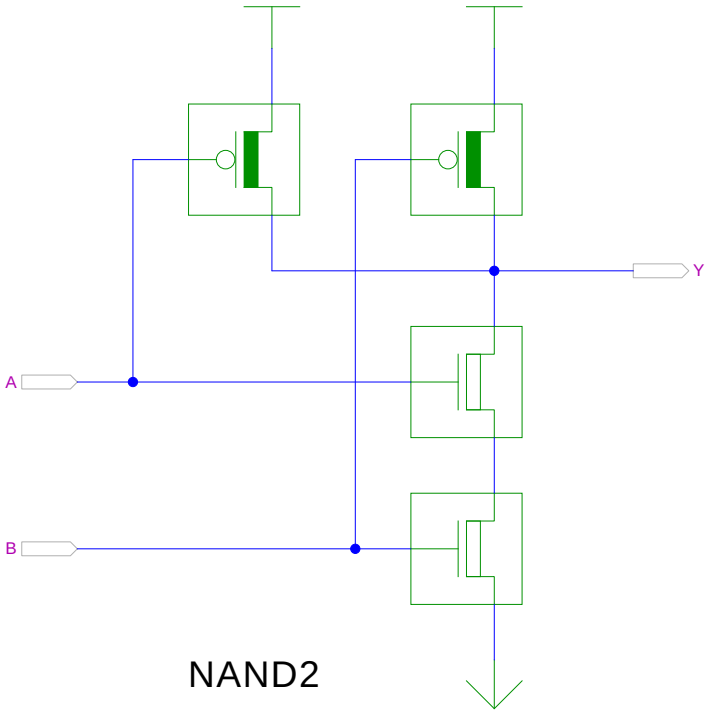




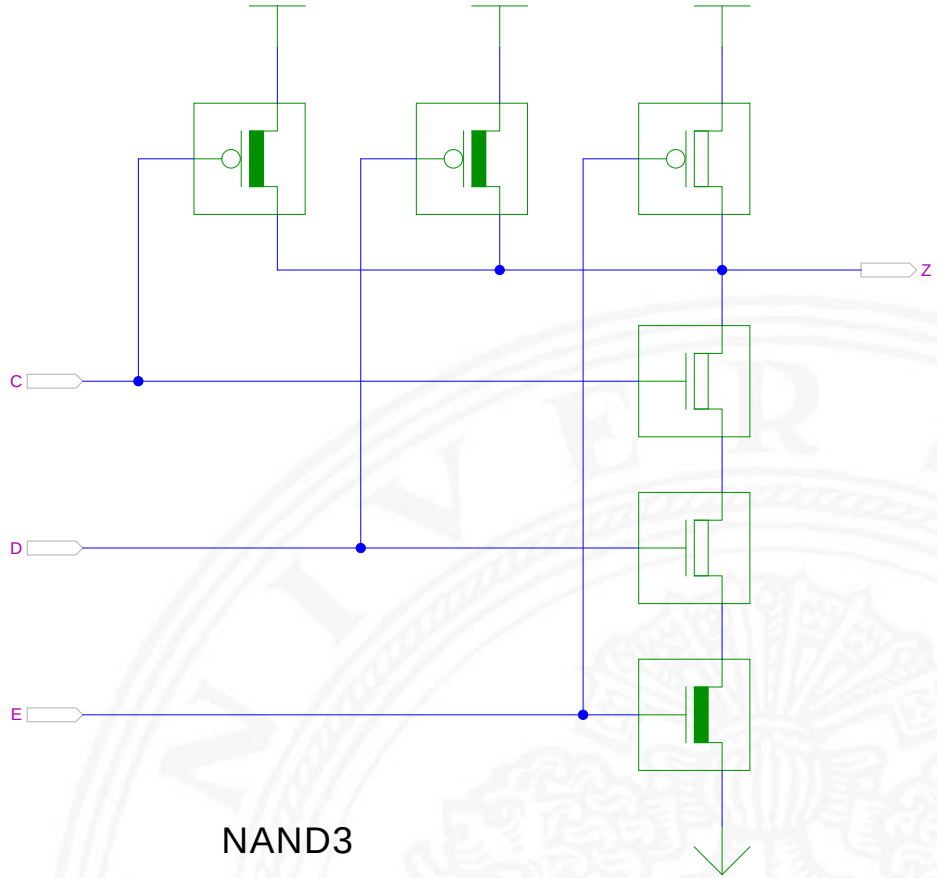
[HenHA] Hades Demo: `05-switched/40-cmos/nand`

- NAND: n-Transistoren in Reihe, p-Transistoren parallel
- AND: Kaskade aus NAND und Inverter

NAND- und AND-Gatter (cont.)



NAND2

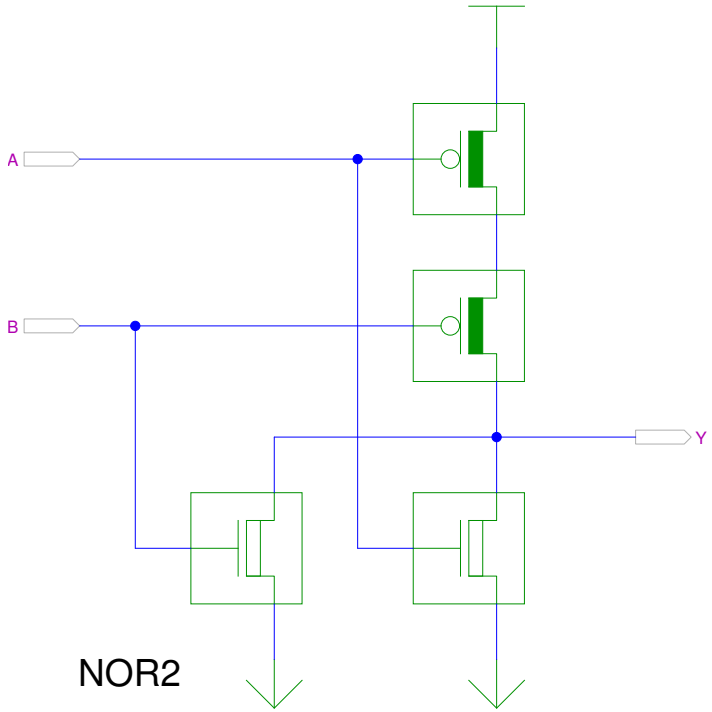


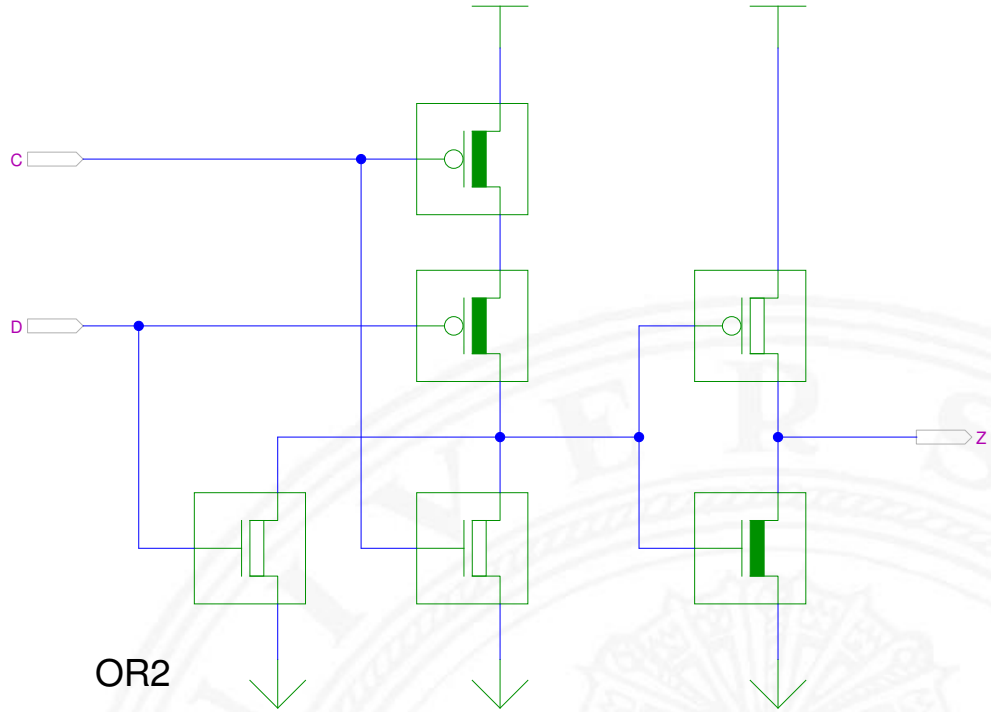
NAND3

[HenHA] Hades Demo: `05-switched/40-cmos/nand3`

- n-Transistoren in Reihe, p-Transistoren parallel
- normalerweise max. 4 Transistoren in Reihe (Spannungsabfall)

NOR- und OR-Gatter





[HenHA] Hades Demo: `05-switched/40-cmos/nor`

- Struktur komplementär zum NAND/AND
- n-Transistoren parallel, p-Transistoren in Reihe
- Reihenschaltung von p-Kanal Transistoren schlechter
  $\Rightarrow$ oft langsamer als NAND

– Ende –

How to make systems energy efficient: Fundamentals of dynamic voltage scaling (DVS)

Power consumption of CMOS circuits (ignoring leakage):

$P = \alpha \, C_L \, V_{dd}^{\,2} \, f$ with

α : switching activity

C_L : load capacitance

V_{dd} : supply voltage

f : clock frequency

Delay for CMOS circuits:

$$\tau = k \, C_L \frac{V_{dd}}{(V_{dd} - V_t)^2}$$ with

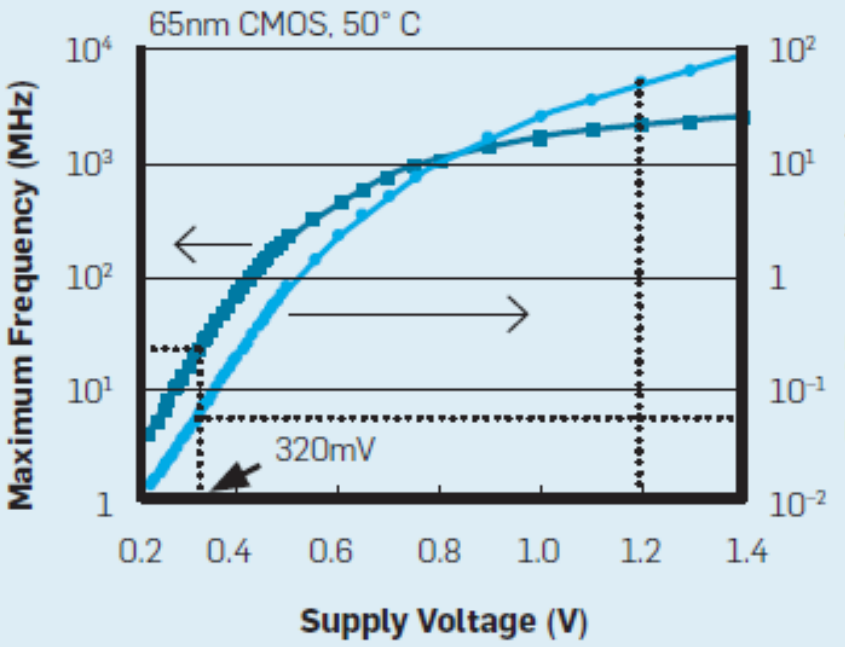
V_t : threshhold voltage

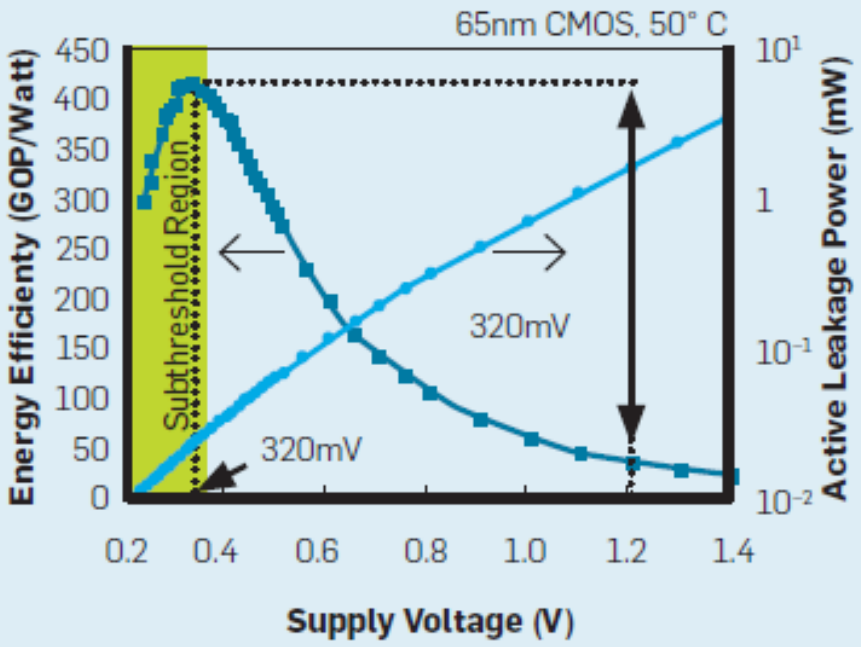
$V_t < V_{dd}$

☞ Decreasing V_{dd} reduces P quadratically, while the run-time of algorithms is only linearly increased

Voltage scaling: Example

Figure 13. Improving energy efficiency through voltage scaling.

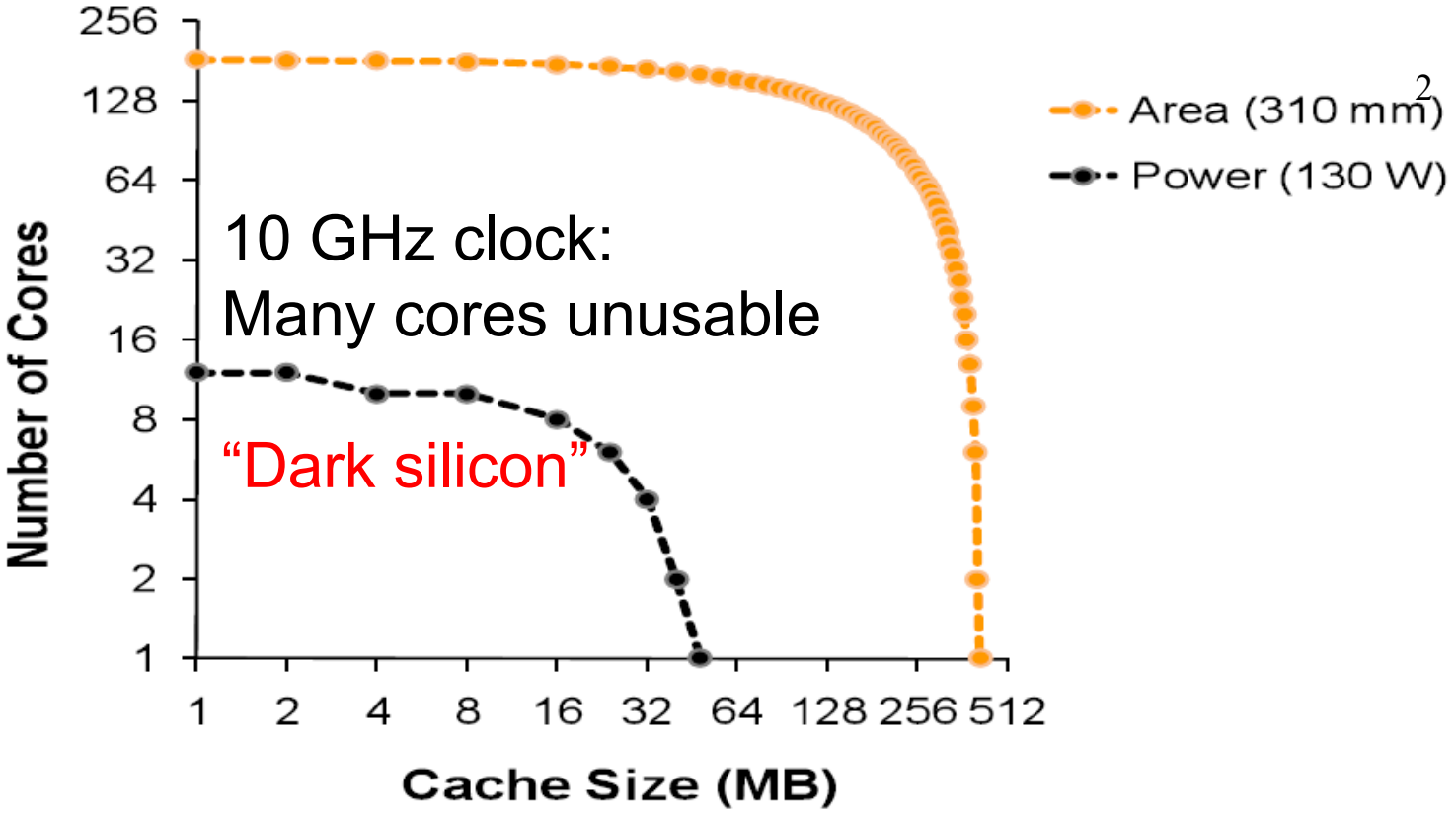




S. Borkar, A. Chien: The future of microprocessors, *Communications of the ACM*, May 2011

Area vs. Power Envelope (22nm)

(for servers)

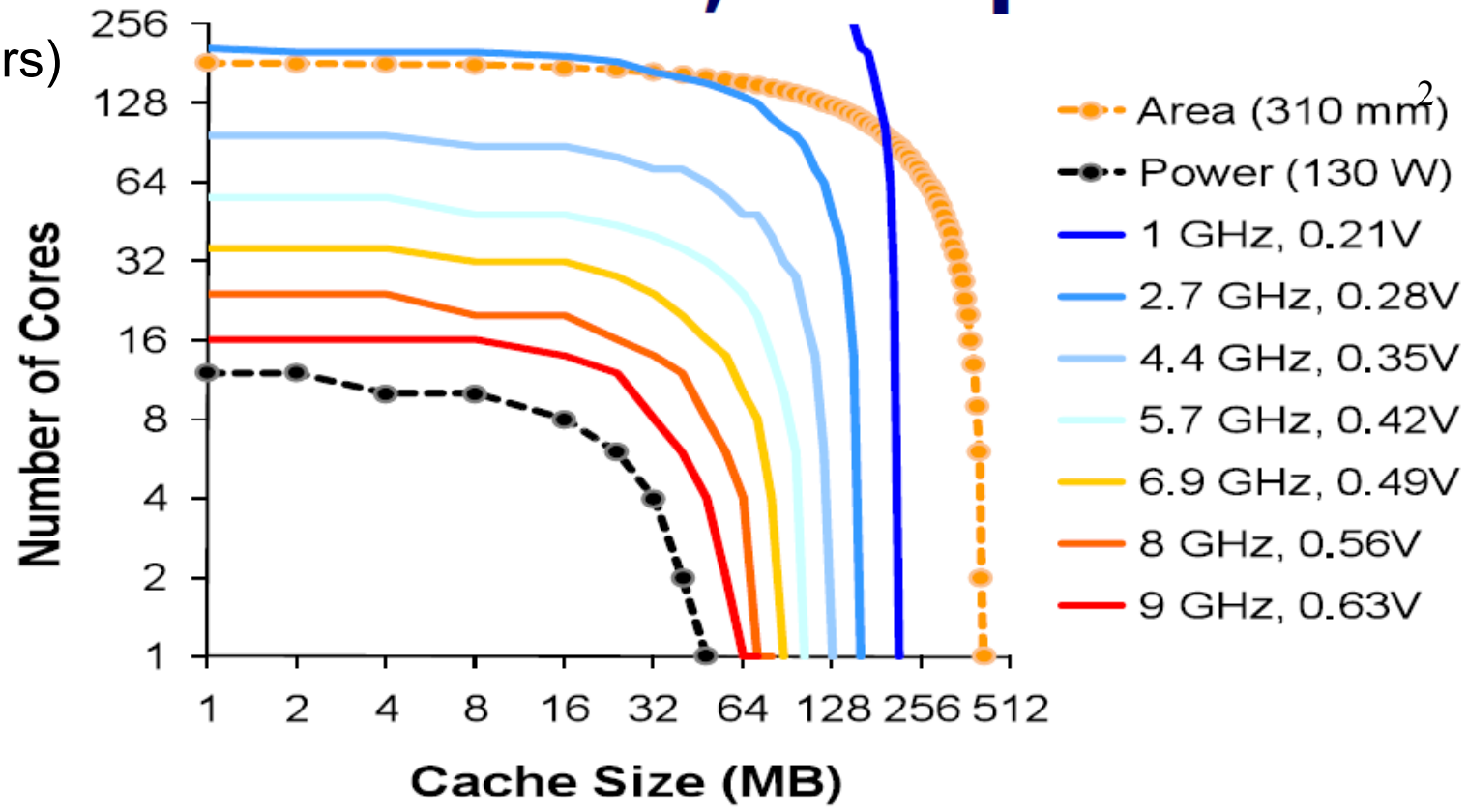


✓ Good news: can fit hundreds of cores

× Can not use them all at highest speed

Of course one could pack more slower cores, cheaper cache

(for servers)

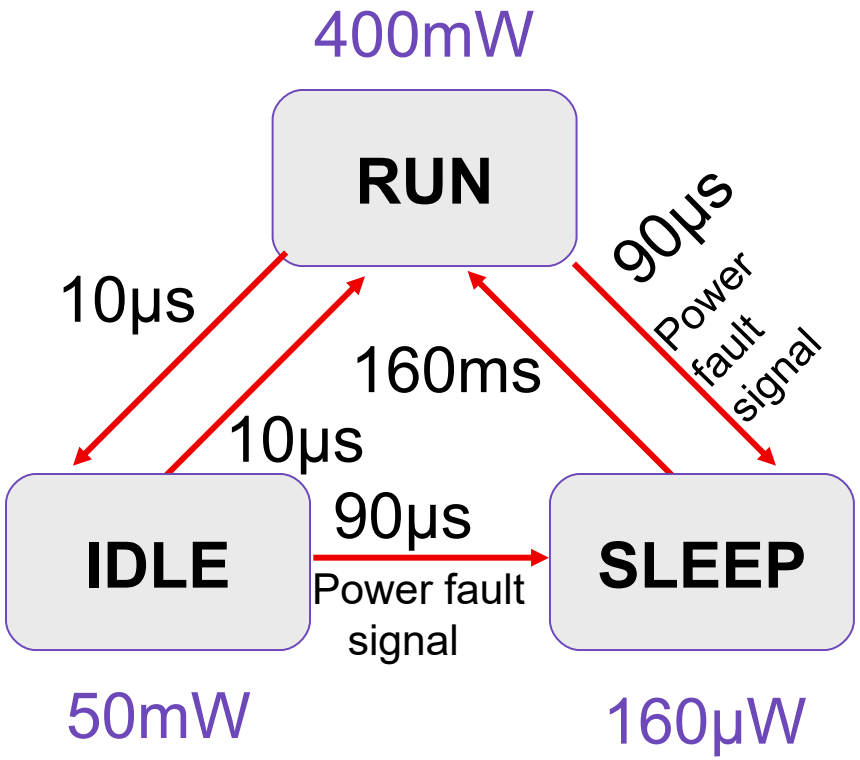


- Result: a performance/power trade-off
- Assuming bandwidth is unlimited

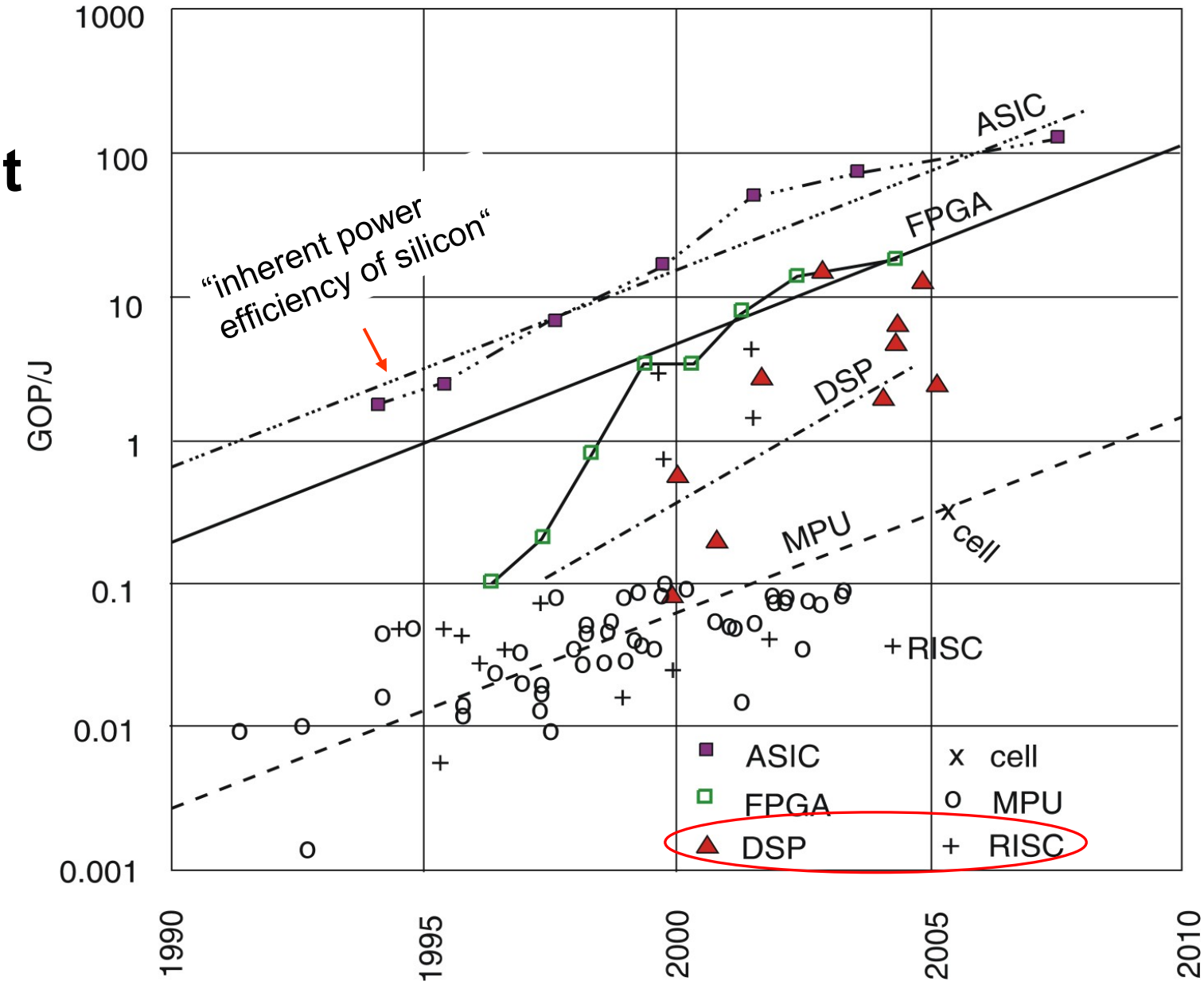
Dynamic power management (DPM)

Example: STRONGARM SA1100

RUN: operational
IDLE: a SW routine may stop the CPU when not in use, while monitoring interrupts
SLEEP: Shutdown of on-chip activity



Energy Efficiency of different target platforms



Low voltage, parallel operation more efficient than high voltage, sequential operation

Basic equations

Power:	$P \sim V_{dd}^2$,
Maximum clock frequency:	$f \sim V_{dd}$,
Energy to run a program:	$E = P \times t$, with: t = runtime (fixed)
Time to run a program:	$t \sim 1 / f$

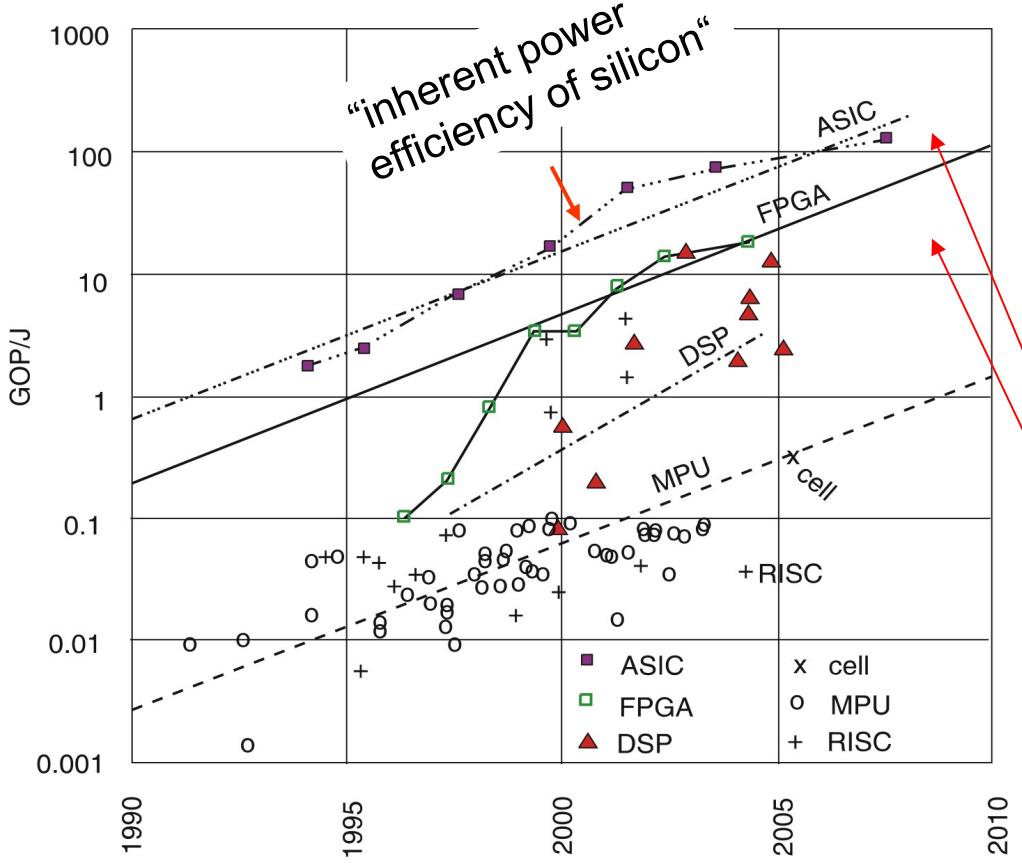
Changes due to parallel processing, with β operations per clock:

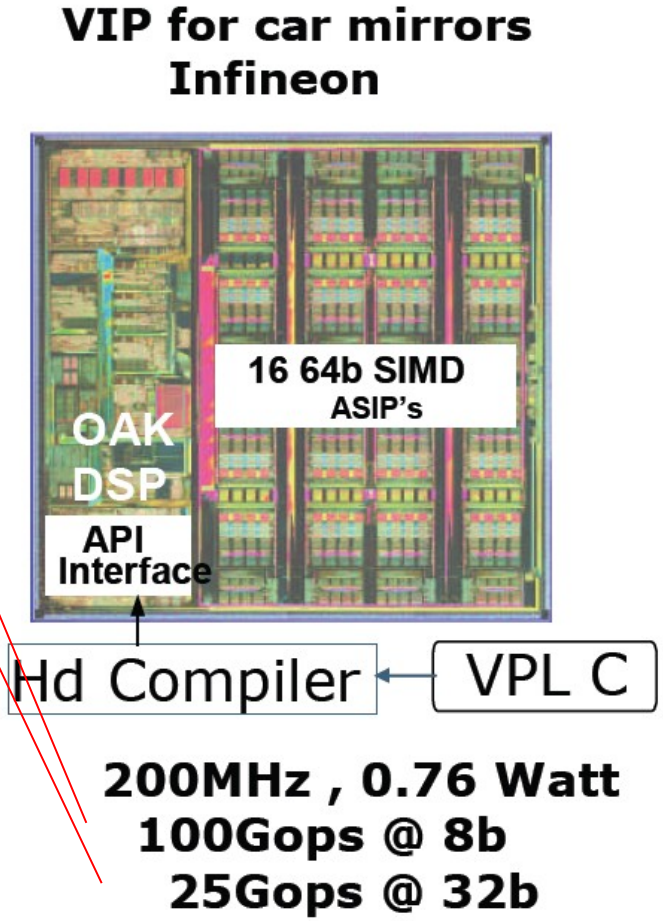
Clock frequency reduced to:	$f' = f / \beta$,
Voltage can be reduced to:	$V_{dd}' = V_{dd} / \beta$,
Power for parallel processing:	$P° = P / \beta^2$ per operation,
Power for β operations per clock:	$P' = \beta \times P° = P / \beta$,
Time to run a program is still:	$t' = t$,
Energy required to run program:	$E' = P' \times t = E / \beta$

☞ Argument in favour of voltage scaling, and parallel processing

Rough approximations!

Energy-efficient architectures: Domain- and application specific (1)

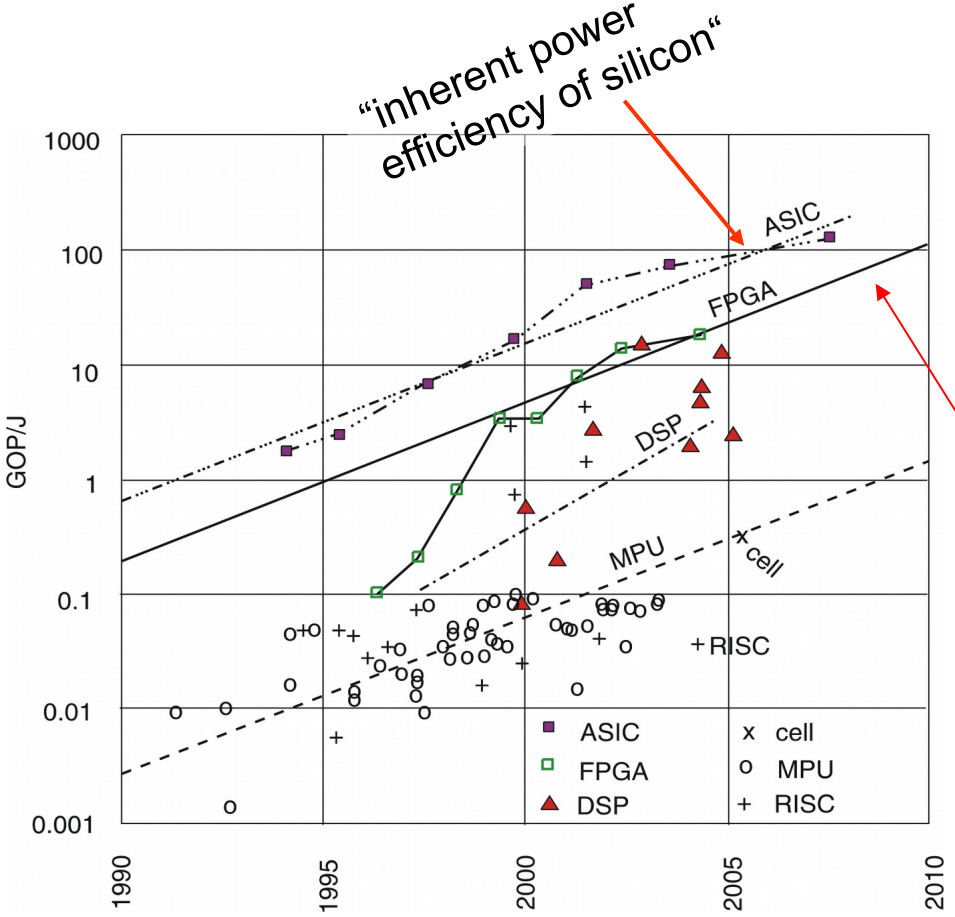


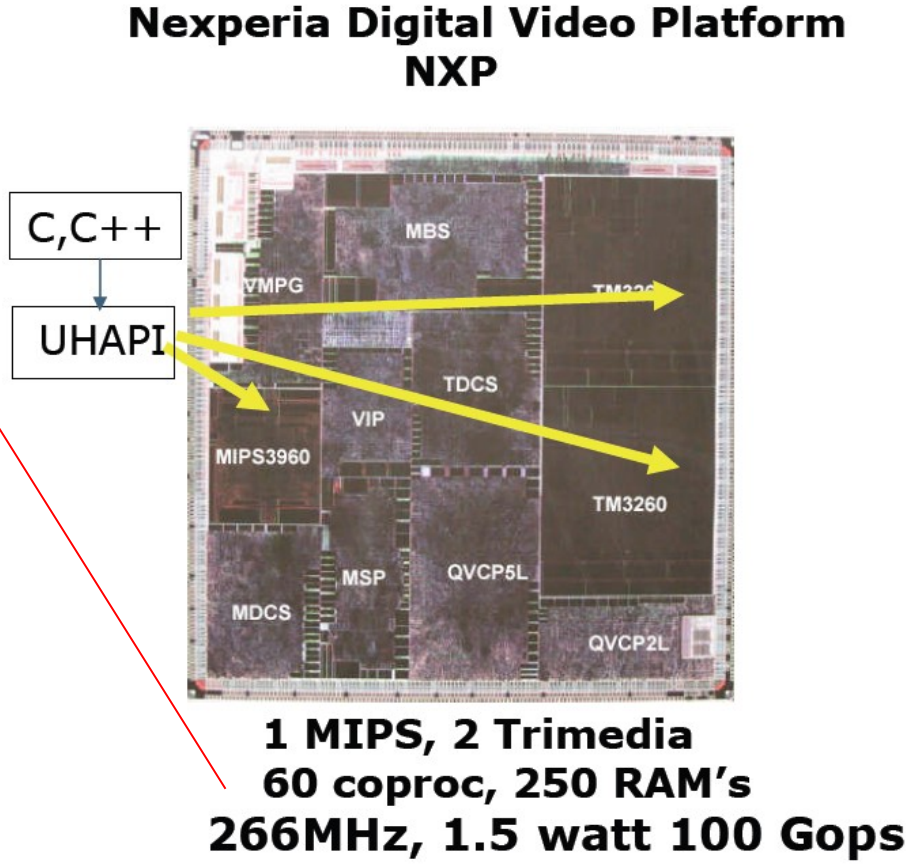


Close to power efficiency of silicon

© Hugo De Man: From the Heaven of Software to the Hell of Nanoscale Physics: An Industry in Transition, Keynote Slides, ACACES, 2007

Energy-efficient architectures: Domain- and application specific (2)



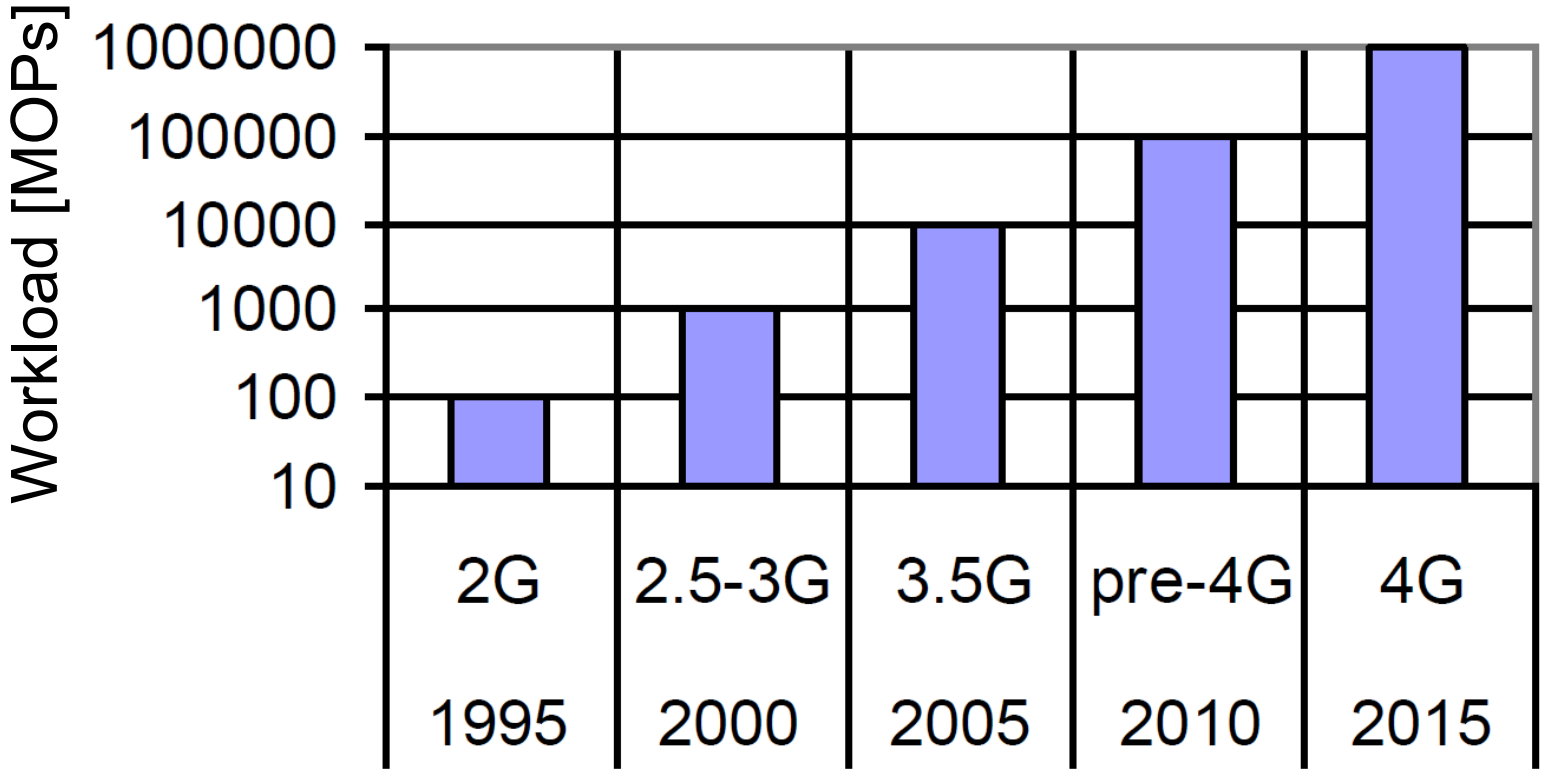


© Hugo De Man: From the Heaven of Software to the Hell of Nanoscale Physics: An Industry in Transition, Keynote Slides, ACACES, 2007

Close to power efficiency of silicon

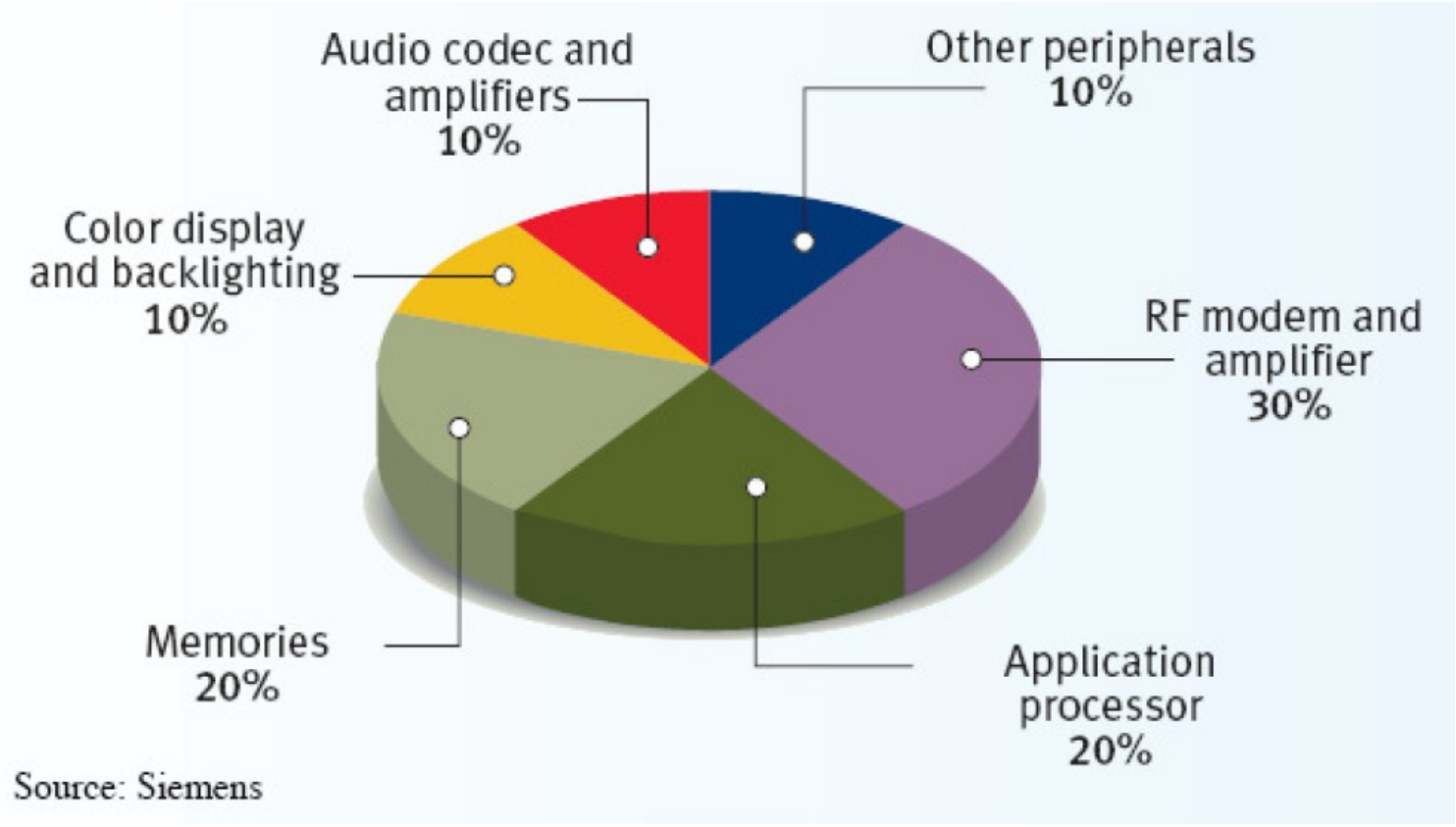
Problem: Increasing performance requirements for mobile phones

C.H. van Berkel: Multi-Core for Mobile Phones, DATE, 2009



Many more instances of the power/energy problem

Where does the power go ?
- mobile phone -

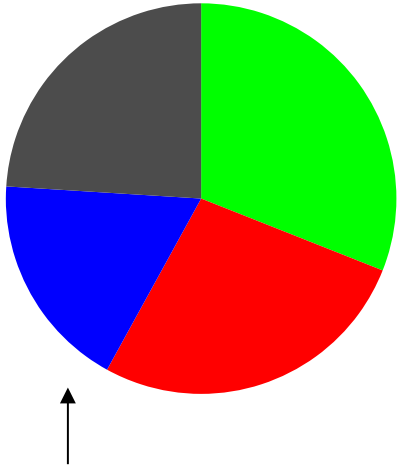


- **It not just I/O, don't ignore processing!**

O. Vargas: Minimum power consumption in mobile-phone memory subsystems; Pennwell Portable Design - September 2005

Where does the power go ? (2)
- Consumer portable systems –

Mobile phone use, breakdown by type of computation



(geometry processing, rasterization, pixel shading)

(display & camera processing, video (de)coding)

(front-end, demodulation, decoding, protocol)

(user interface, browsing, …)

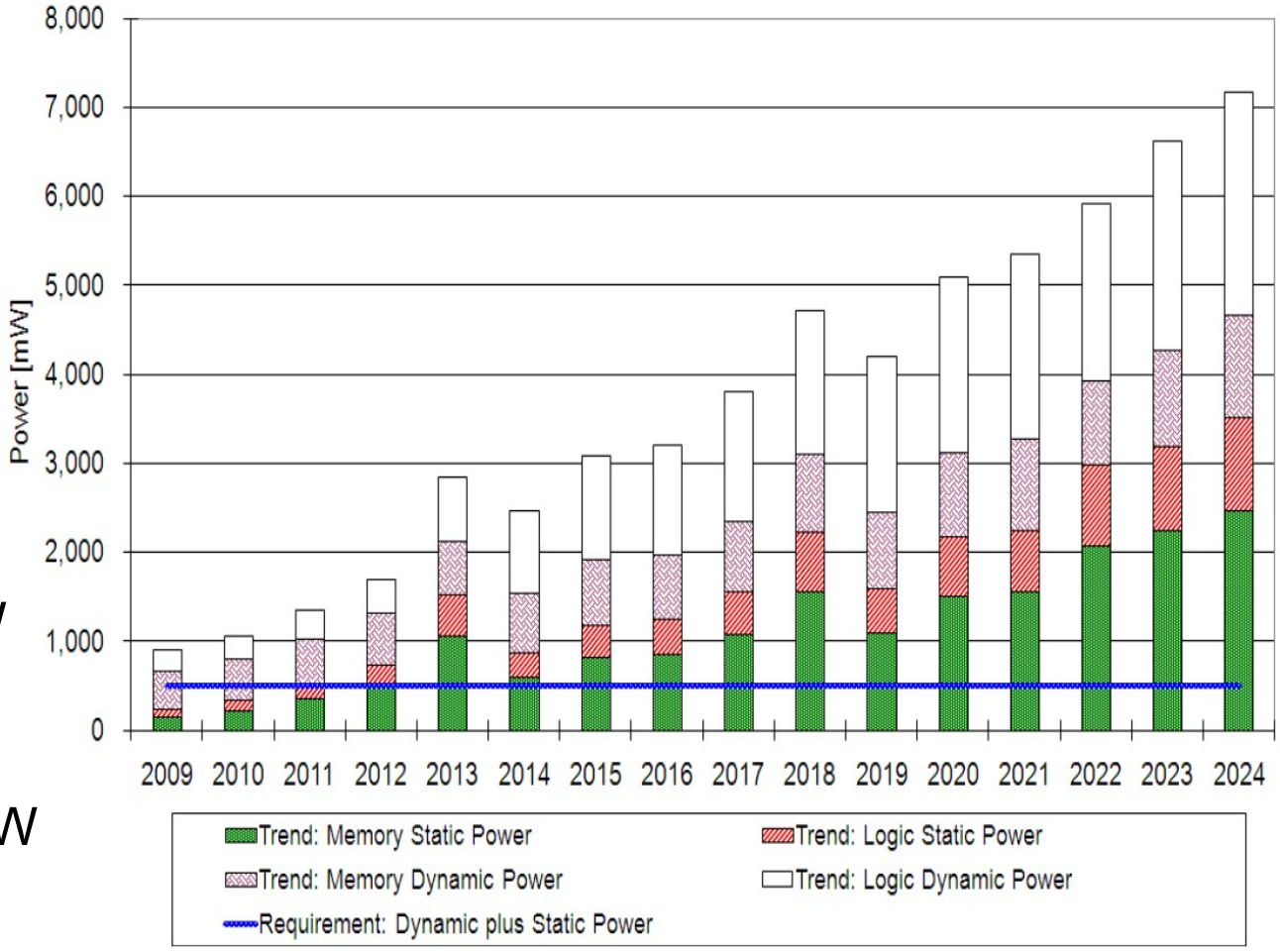
With special purpose HW!

C.H. van Berkel: Multi-Core for Mobile Phones, DATE, 2009
(no explicit percentages in original paper)

☞ During use, all components & computations relevant

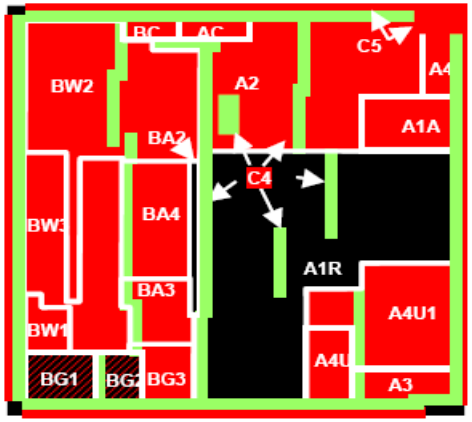
Where is the energy consumed ?
- Consumer portable systems -

- According to *International Technology Roadmap for Semiconductors* (ITRS), 2010 update [www.itrs.net]
- Current trends
  - violation of 0.5-1 W constraint for small mobiles
  - large mobiles: ~ 7 W



Energy-efficient architectures: Heterogeneous processors

(2)Telephony (W-CDMA)



Power on

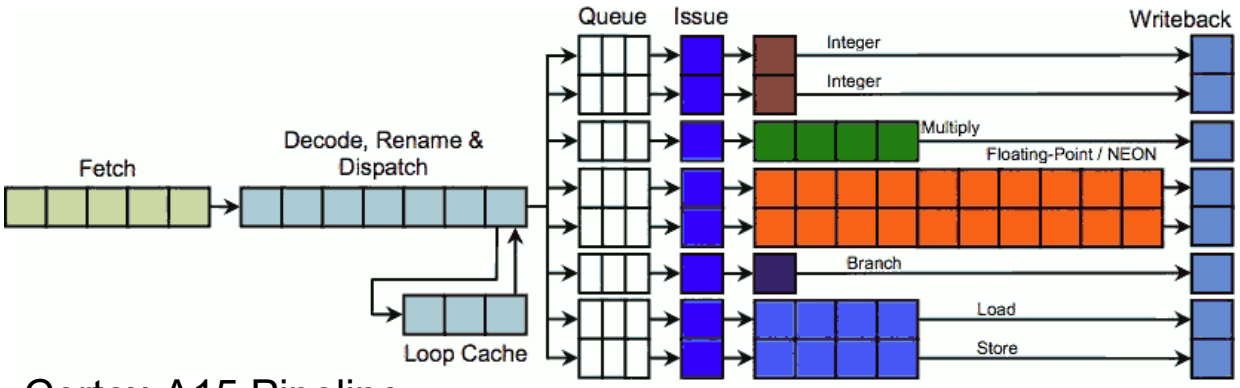
Power off

Baseband part	Control	ON
	W-CDMA	ON
	GSM	ON / OFF
Application part	System-domain	ON
	Realtime-domain	OFF
Measured Leakage Current (@ Room Temp, 1.2V)		407 μA

http://www.mpsoc-forum.org/2007/slides/Hattori.pdf

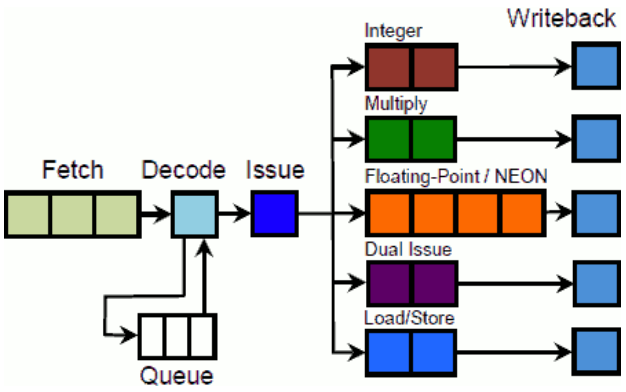
☞ ***"Dark silicon"*** (not all silicon can be powered at the same time, due to current, power or temperature constraints)

ARM‘s big.LITTLE as an example

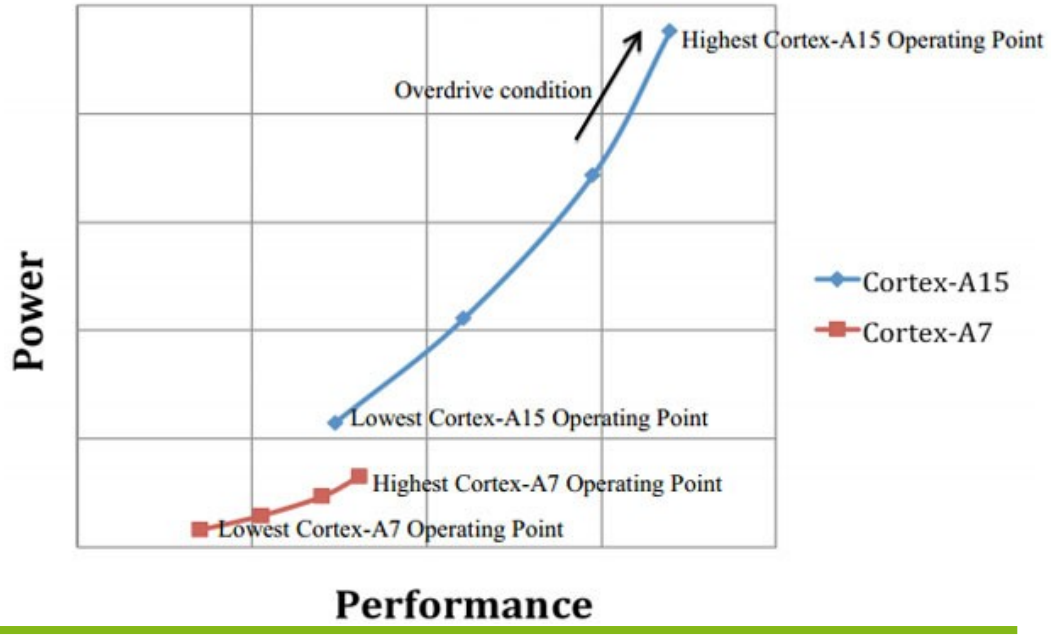


Cortex-A15 Pipeline

Used in Samsung S4

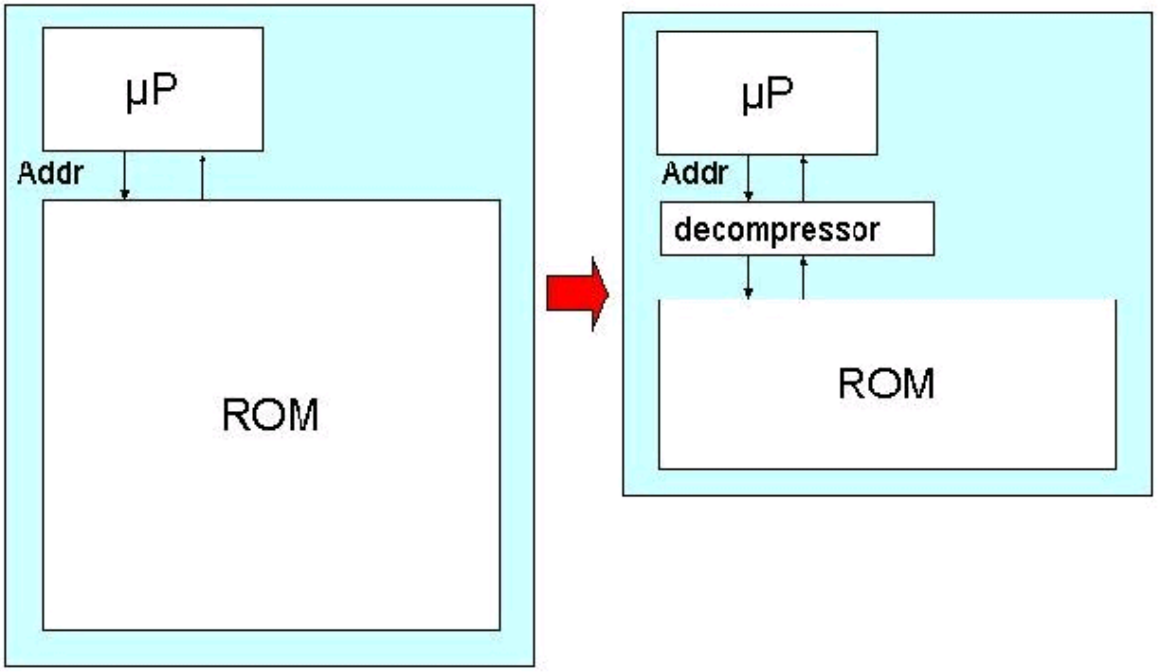


Cortex-A7 Pipeline



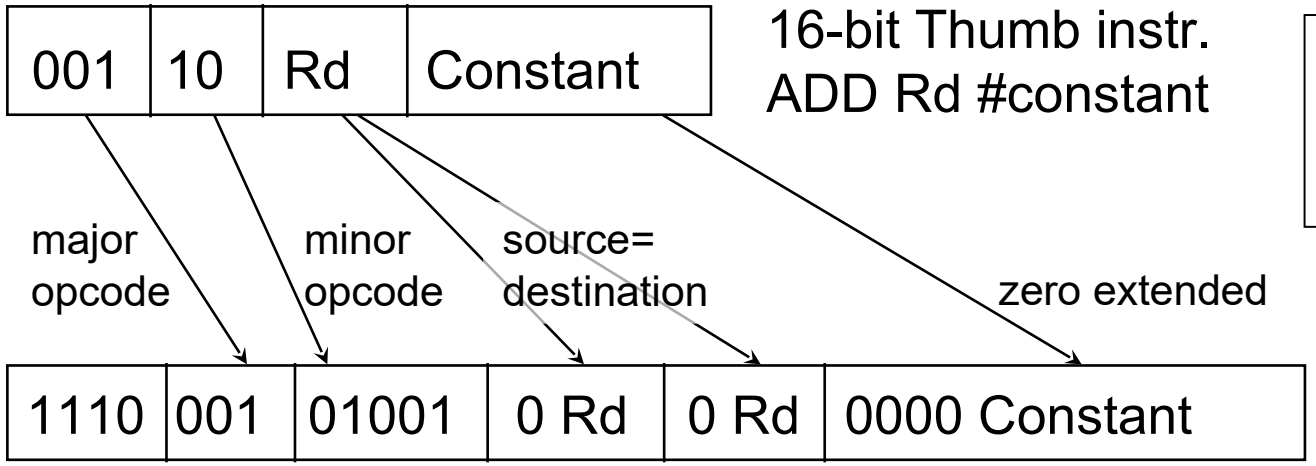
Key requirement: Code-size efficiency

- Overview: http://www-perso.iro.umontreal.ca/~latendre/codeCompression/codeCompression/node1.html
- **Compression techniques:** key idea



Code-size efficiency

- **Compression techniques (continued):**
  - 2nd instruction set, e.g. ARM Thumb instruction set:



  - Reduction to 65-70 % of original code size
  - 130% of ARM performance with 8/16 bit memory
  - 85% of ARM performance with 32-bit memory

  - Same approach for LSI TinyRisc, …
  - Requires support by compiler, assembler etc.

Dictionary approach, two level control store (indirect addressing of instructions)

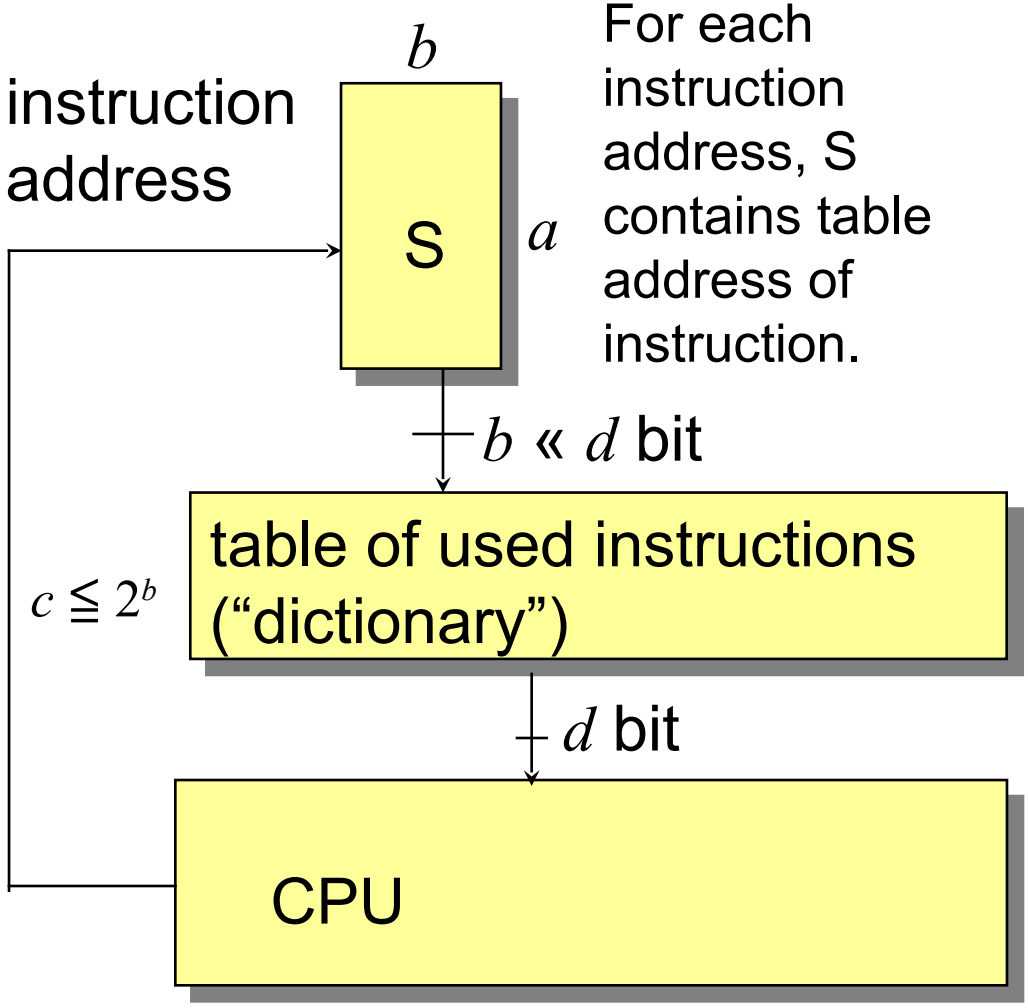
"*Dictionary-based coding schemes cover a wide range of various coders and compressors.*

Their common feature is that the methods use some kind of a dictionary that contains parts of the input sequence which frequently appear.

The encoded sequence in turn contains references to the dictionary elements rather than containing these over and over."

[Á. Beszédes et al.: Survey of Code size Reduction Methods, Survey of Code-Size Reduction Methods, *ACM Computing Surveys*, Vol. 35, Sept. 2003, pp 223-267]

Key idea (for d bit instructions)



Uncompressed storage of a d-bit-wide instructions requires a x d bits.

In compressed code, each instruction pattern is stored only once.

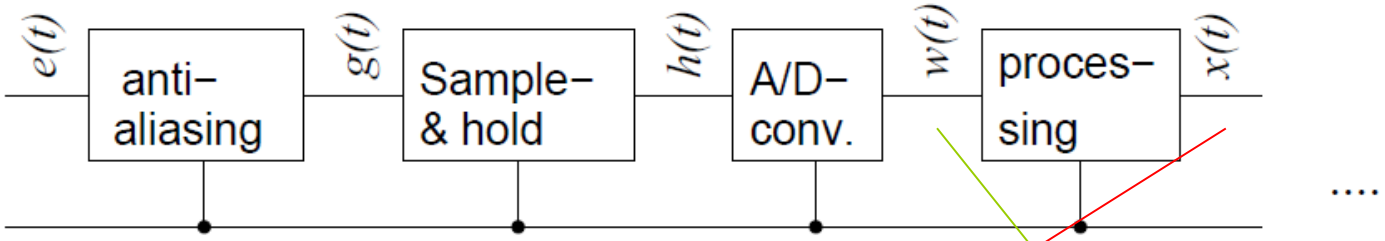
small

Hopefully, axb+cxd < axd.

Called nanoprogramming in the Motorola 68000.

Key requirement: Run-time efficiency - Domain-oriented architectures

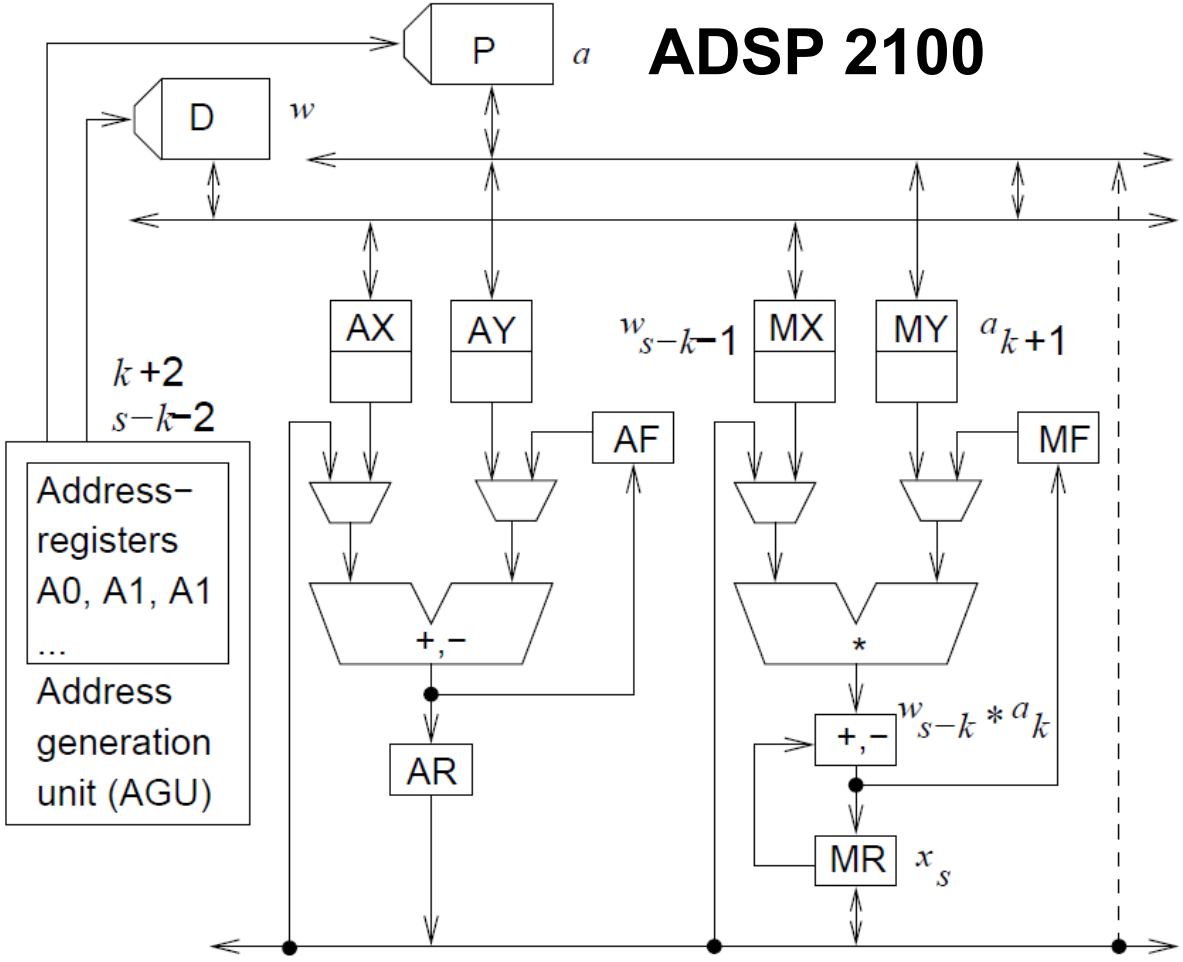
Example: Filtering in Digital signal processing (DSP)



$$x_s = \sum_{k=0}^{n-1} w_{s-k} * a_k$$

Signal at $t = t_s$ (sampling points)

Filtering in digital signal processing



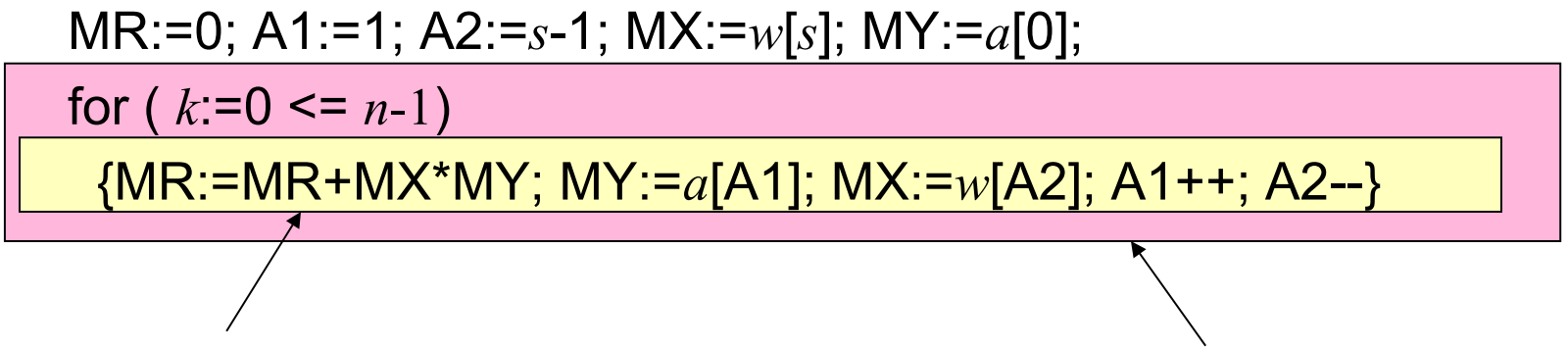
$$x_s = \sum_{k=0}^{n-1} w_{s-k} * a_k$$

-- outer loop over
-- sampling times t_s

```
{ MR:=0; A1:=1; A2:=s-1;
  MX:=w[s]; MY:=a[0];
  for (k=0; k <= (n−1); k++)
  { MR:=MR + MX * MY;
     MX:=w[A2]; MY:=a[A1];
     A1++; A2--;
  }
  x[s]:=MR;
}
```

- Maps nicely

DSP-Processors: multiply/accumulate (MAC) and zero-overhead loop (ZOL) instructions

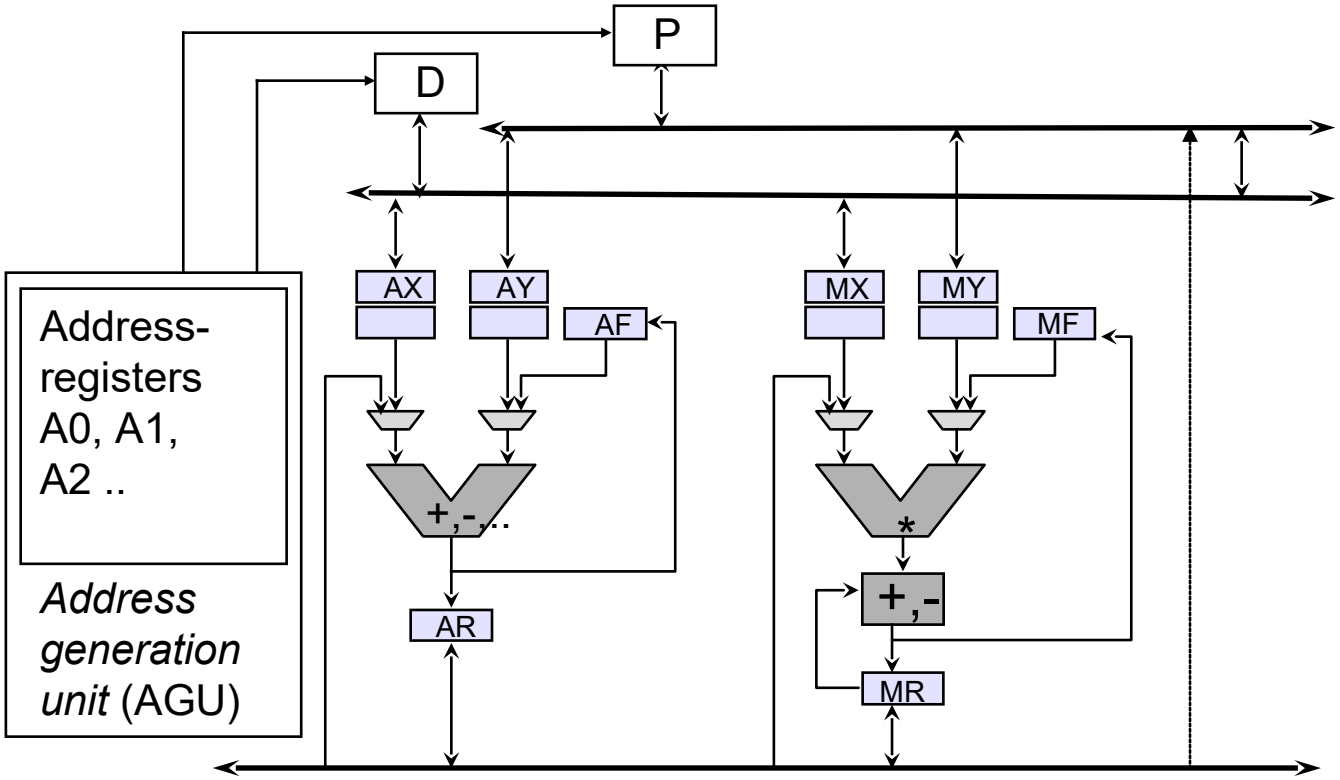


Multiply/accumulate (MAC) instruction

Zero-overhead loop (ZOL) instruction preceding MAC instruction.
Loop testing done in parallel to MAC operations.

Heterogeneous registers

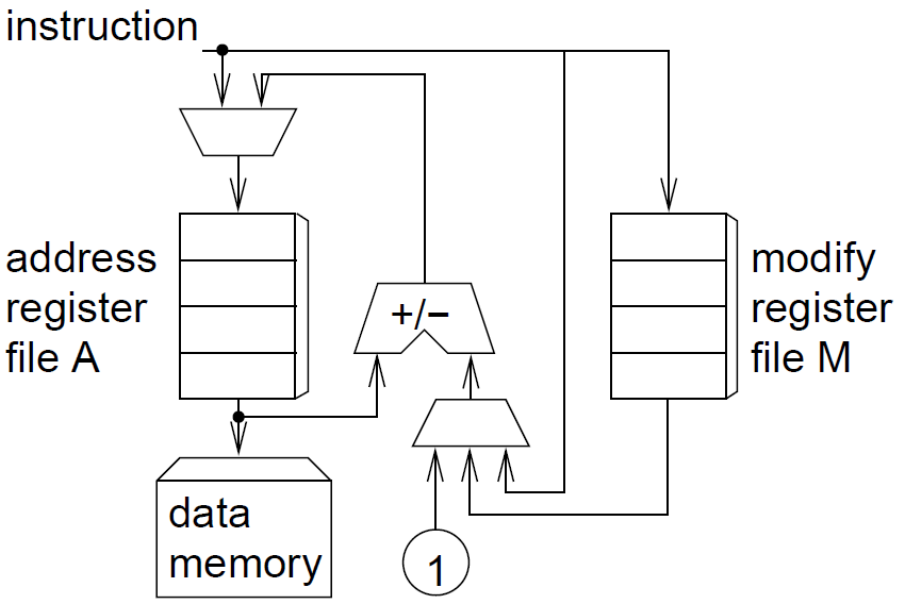
Example (ADSP 210x):



Different functionality of registers An, AX, AY, AF, MX, MY, MF, MR

Separate address generation units (AGUs)

Example (ADSP 210x):



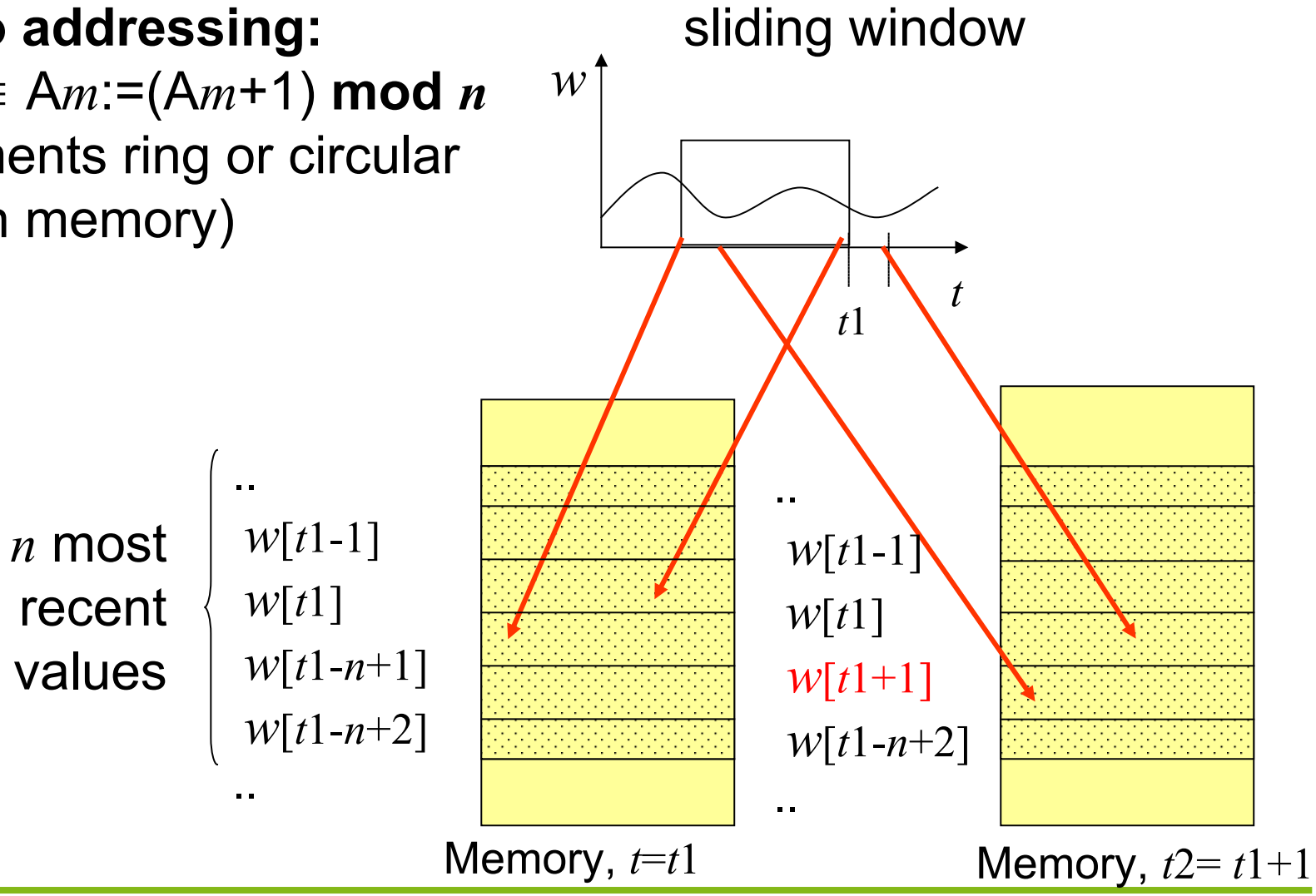
- Data memory can only be fetched with address contained in A,
- but this can be done in parallel with operation in main data path (takes effectively 0 time).
- A := A ± 1 also takes 0 time,
- same for A := A ± M;
- A := <immediate in instruction> requires extra instruction

☞ Minimize load immediates

☞ Optimization in optimization chapter

Modulo addressing

Modulo addressing:
Am++ ≡ Am:=(Am+1) **mod** n
(implements ring or circular buffer in memory)



Saturating arithmetic

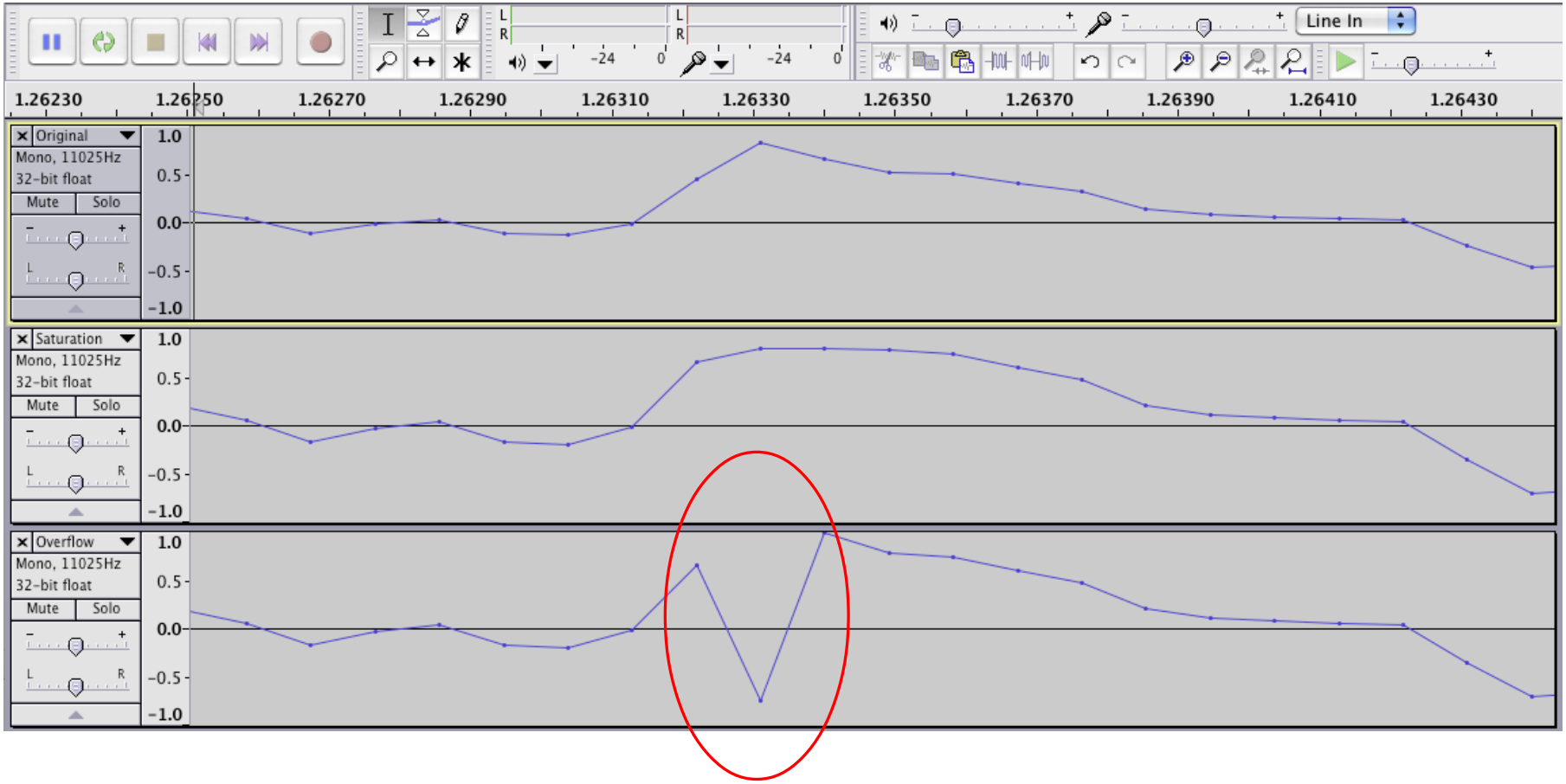
- Returns largest/smallest number in case of over/underflows
- Example:

a			0111
b		+	1001
standard wrap around arithmetic			(1)0000
saturating arithmetic			1111
(a+b)/2:	correct		1000
	wrap around arithmetic		0000
	saturating arithmetic + shifted		0111

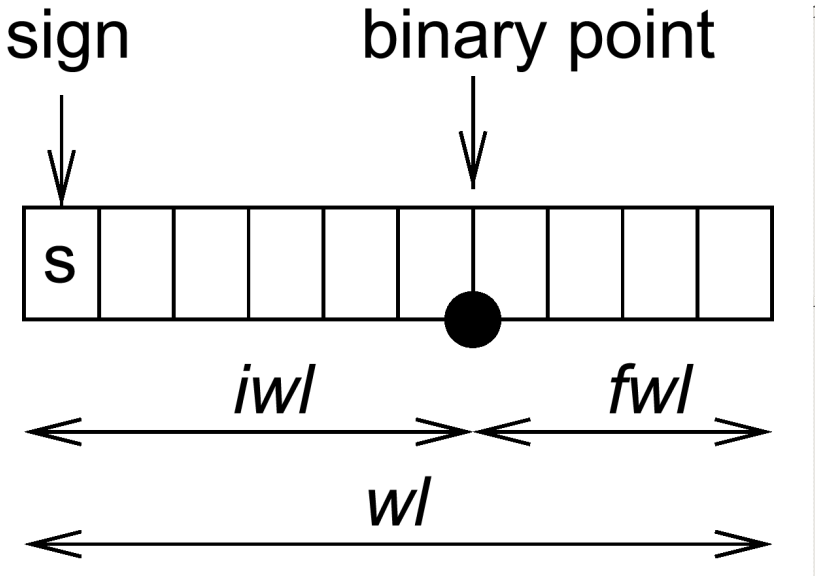
"almost correct"

- Appropriate for DSP/multimedia applications:
  - No timeliness of results if interrupts are generated for overflows
  - Precise values less important
  - Wrap around arithmetic would be worse.

Example


Line In
1.26230
1.26250
1.26270
1.26290
1.26310
1.26330
1.26350
1.26370
1.26390
1.26410
1.26430
Original
Mono, 11025Hz
32-bit float
Mute
Solo
Saturation
Mono, 11025Hz
32-bit float
Mute
Solo
Overflow
Mono, 11025Hz
32-bit float
Mute
Solo
1.0
0.5
0.0
-0.5
-1.0

Fixed-point arithmetic



Shifting required after multiplications and divisions in order to maintain binary point.

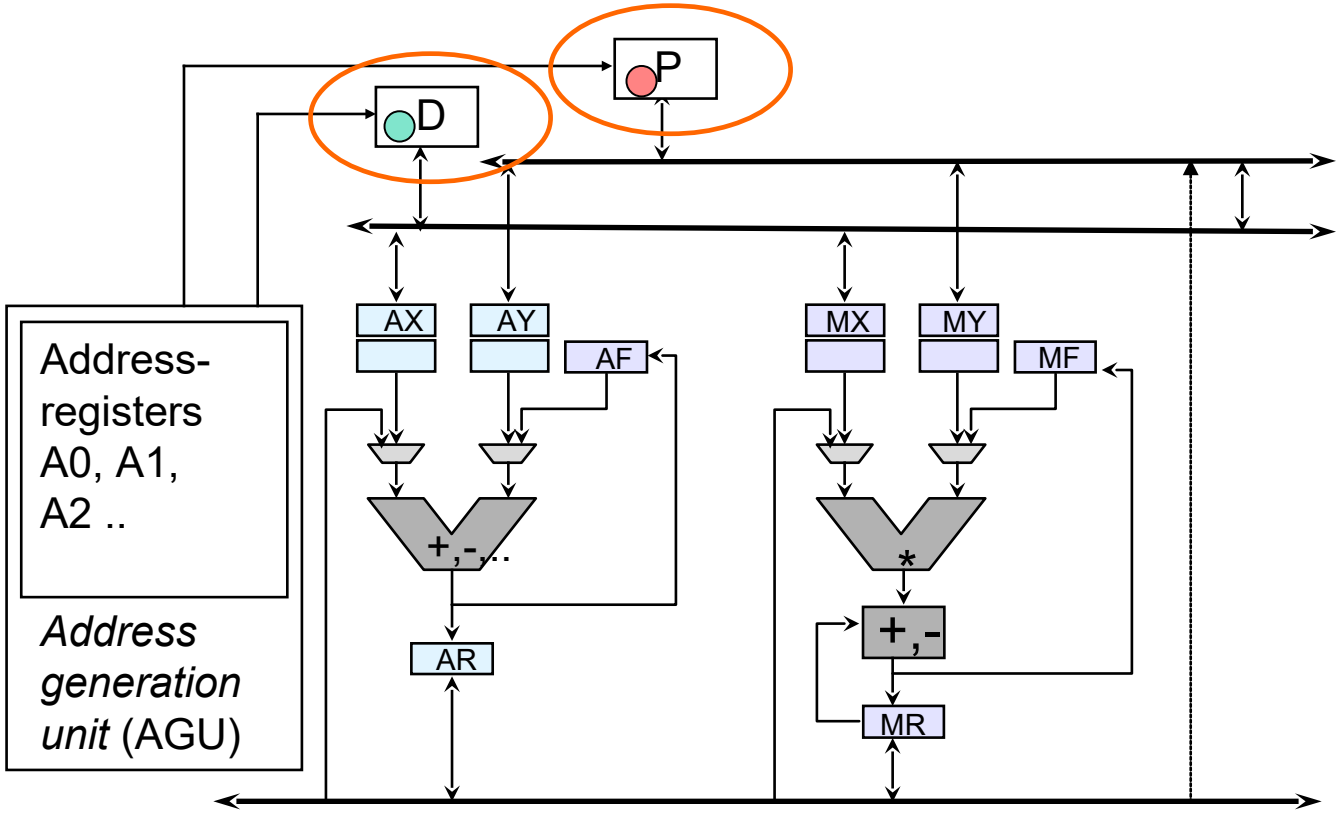
Real-time capability

- **Timing behavior has to be predictable**
  Features that cause problems:
  - Unpredictable access to shared resources
    - Caches with difficult to predict replacement strategies
    - Unified caches (conflicts between instructions and data)
    - Pipelines with difficult to predict stall cycles ("bubbles")
    - Unpredictable communication times for multiprocessors
  - Branch prediction, speculative execution
  - Interrupts that are possible any time
  - Memory refreshes that are possible any time
  - Instructions that have data-dependent execution times

  ☞ Trying to avoid as many of these as possible

Dagstuhl workshop on predictability, Nov. 17-19, 2003

Multiple memory banks or memories

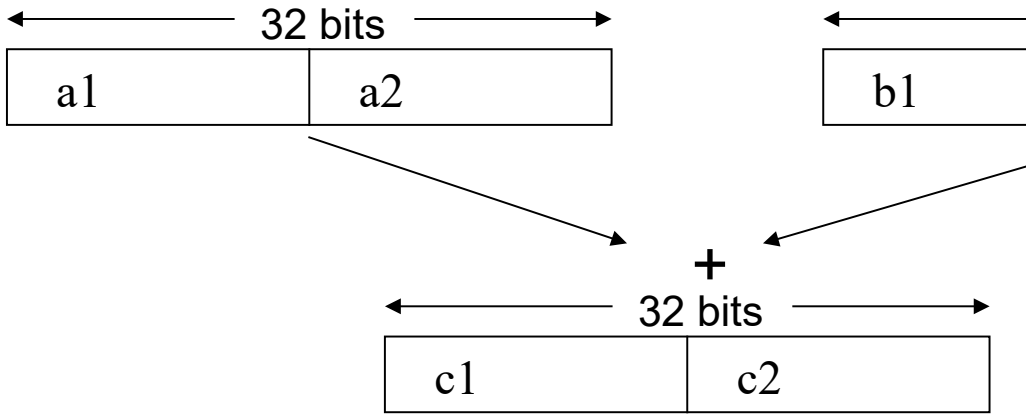


Simplifies parallel fetches

Multimedia-Instructions, Short vector extensions, Streaming extensions, SIMD instructions

- Multimedia instructions exploit that many registers, adders etc are quite wide (32/64 bit), whereas most multimedia data types are narrow

☞ 2-8 values can be stored per register and added. E.g.:



2 additions per instruction; no carry at bit 16

- Cheap way of using parallelism

☞ SSE instruction set extensions, SIMD instructions

Summary

Hardware in a loop

- Sensors
- Discretization
- Information processing
  - Importance of energy efficiency
  - Special purpose HW very expensive
  - Energy efficiency of processors
  - Code size efficiency
  - Run-time efficiency
  - MPSoCs
- D/A converters
- Actuators

technische universität
dortmund

fakultät für informatik
informatik 12

Embedded System Hardware

- Processing -

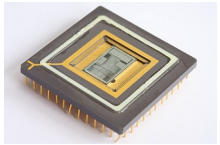
Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 11 月 20 日

Efficiency:
slide from lecture 1 applied to processing

- CPS & ES must be **efficient**

- Code-size efficient
  (especially for systems on a chip)



- Run-time efficient

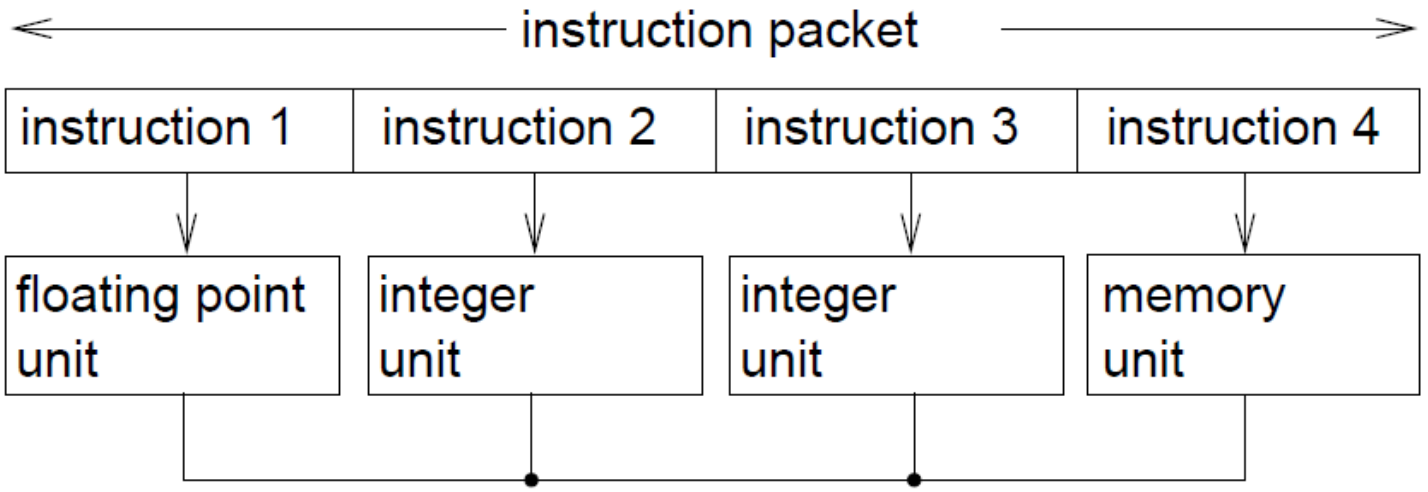
- Weight efficient

- Cost efficient

- Energy efficient

Key idea of very long instruction word (VLIW) computers (1)

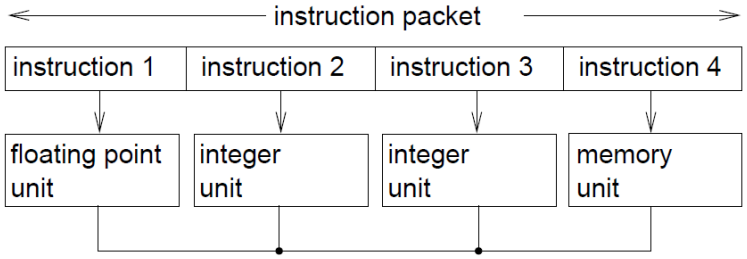
- Instructions included in long instruction packets.
- Instruction packets are assumed to be executed in parallel.
- Fixed association of packet bits with functional units.



- Compiler is assumed to generate these “parallel” packets

Key idea of very long instruction word (VLIW) computers (2)

- Complexity of finding parallelism is moved from the hardware (RISC/CISC processors) to the compiler;



- Ideally, this avoids the overhead (silicon, energy, ..) of identifying parallelism at run-time.

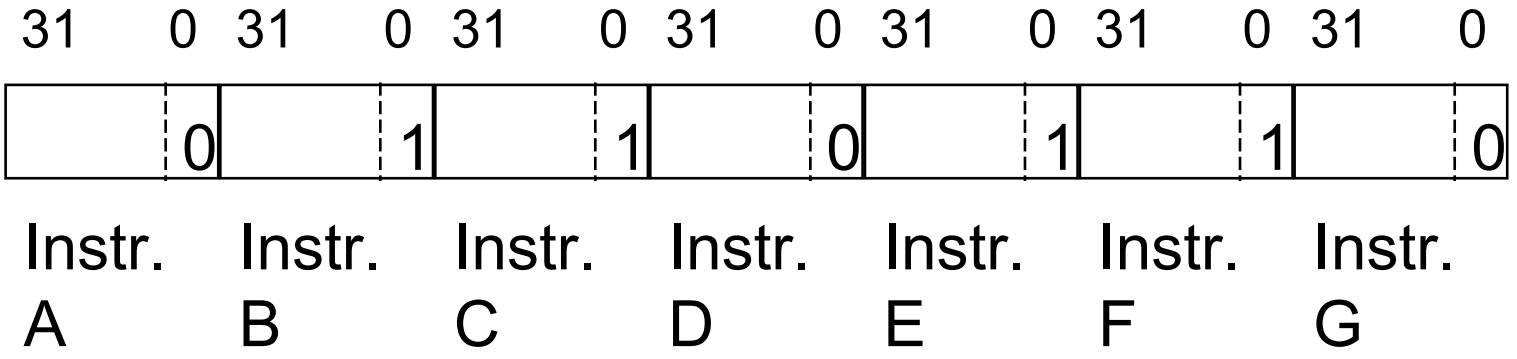
☞ A lot of expectations into VLIW machines

- However, possibly low code efficiency, due to many NOPs

☞ Explicitly parallel instruction set computers (EPICs) are an extension of VLIW architectures: parallelism detected by compiler, but no need to encode parallelism in 1 word.

EPIC: TMS 320C6xxx as an example

1 Bit per instruction encodes end of parallel exec.



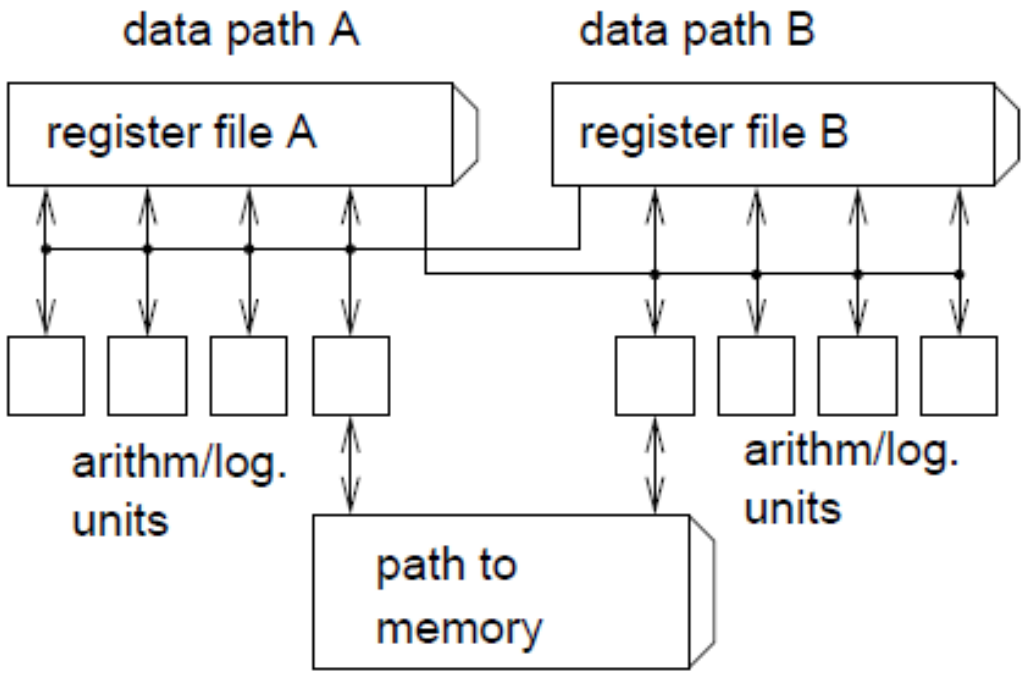
Cycle	Instruction		
1	A		
2	B	C	D
3	E	F	G

Instructions B, C and D use disjoint functional units, cross paths and other data path resources. The same is also true for E, F and G.

Partitioned register files

- Many memory ports are required to supply enough operands per cycle.
- Memories with many ports are expensive.

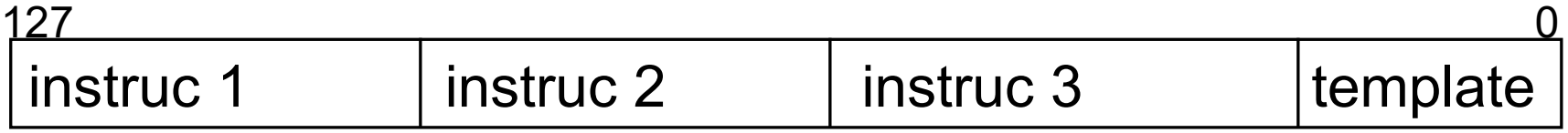
☞ Registers are partitioned into (typically 2) sets, e.g. for TI C6xxx:



Parallel execution cannot span several packets ☞ IA64

More encoding flexibility with IA-64 Itanium

3 instructions per **bundle:**



There are 5 instruction types:

- A: common ALU instructions
- I: more special integer instructions (e.g. shifts)
- M: Memory instructions
- F: floating point instructions
- B: branches

Instruction grouping information

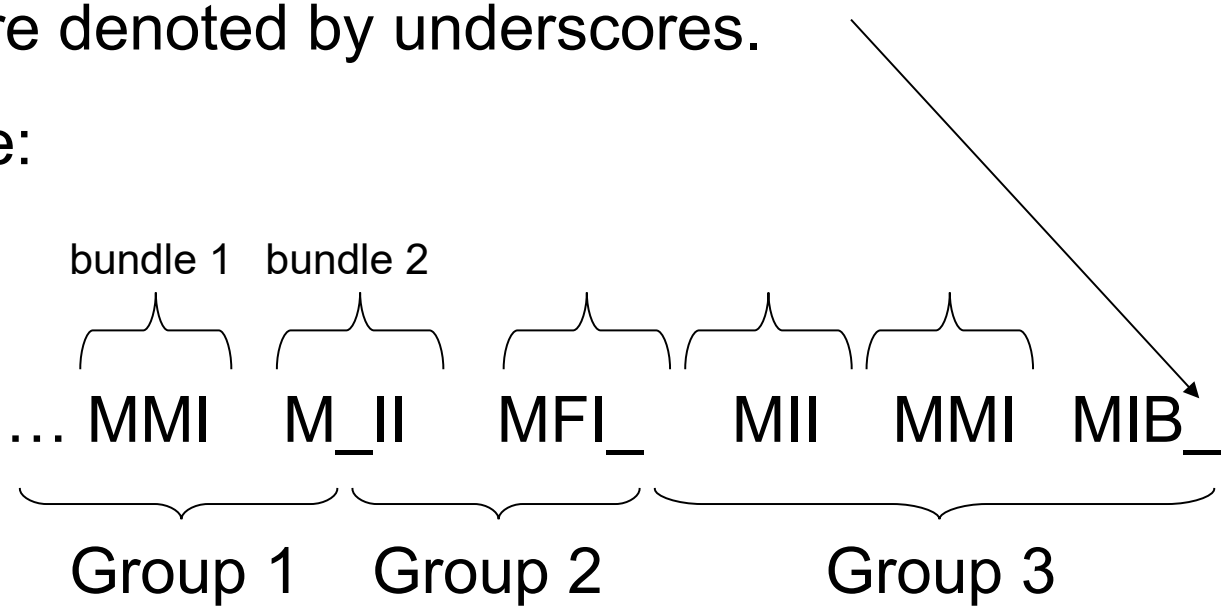
The following combinations can be encoded in templates:

- MII, MMI, MFI, MIB, MMB, MFB, MMF, MBB, BBB, MLX
  with LX = *move 64-bit immediate* encoded in 2 slots

Templates and instruction types

End of parallel execution called **stops.**
Stops are denoted by underscores.

Example:

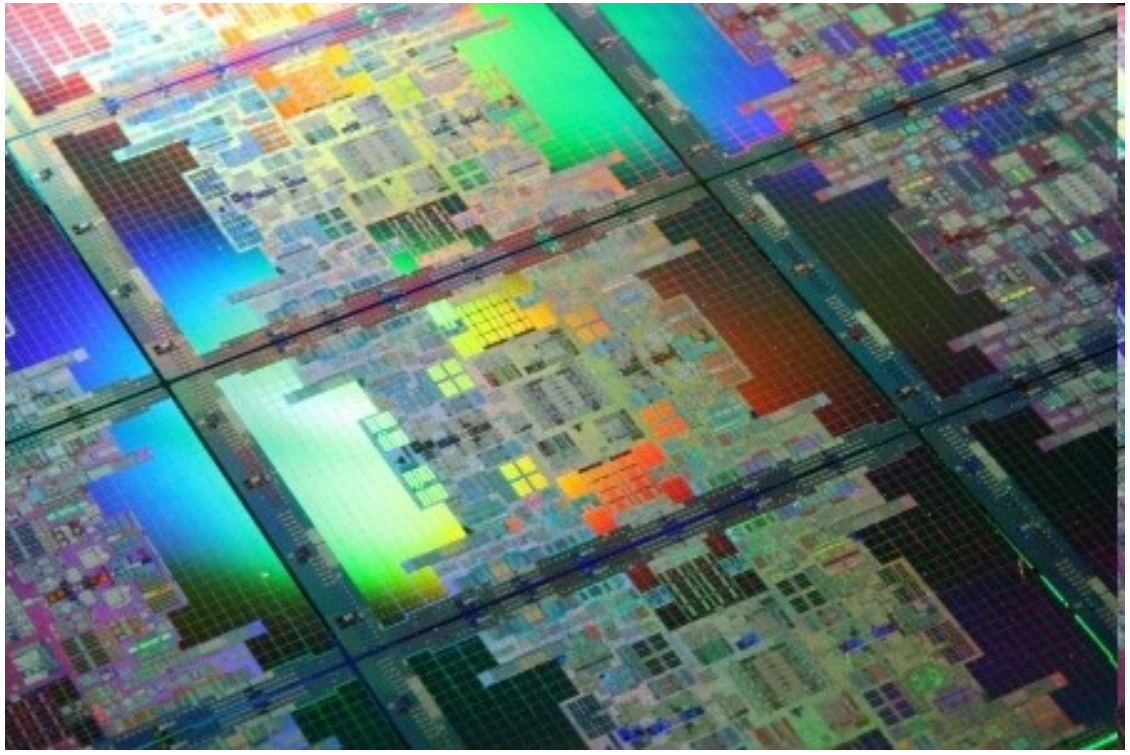


Very restricted placement of stops within bundle.

Parallel execution within groups possible.

Parallel execution can span several bundles

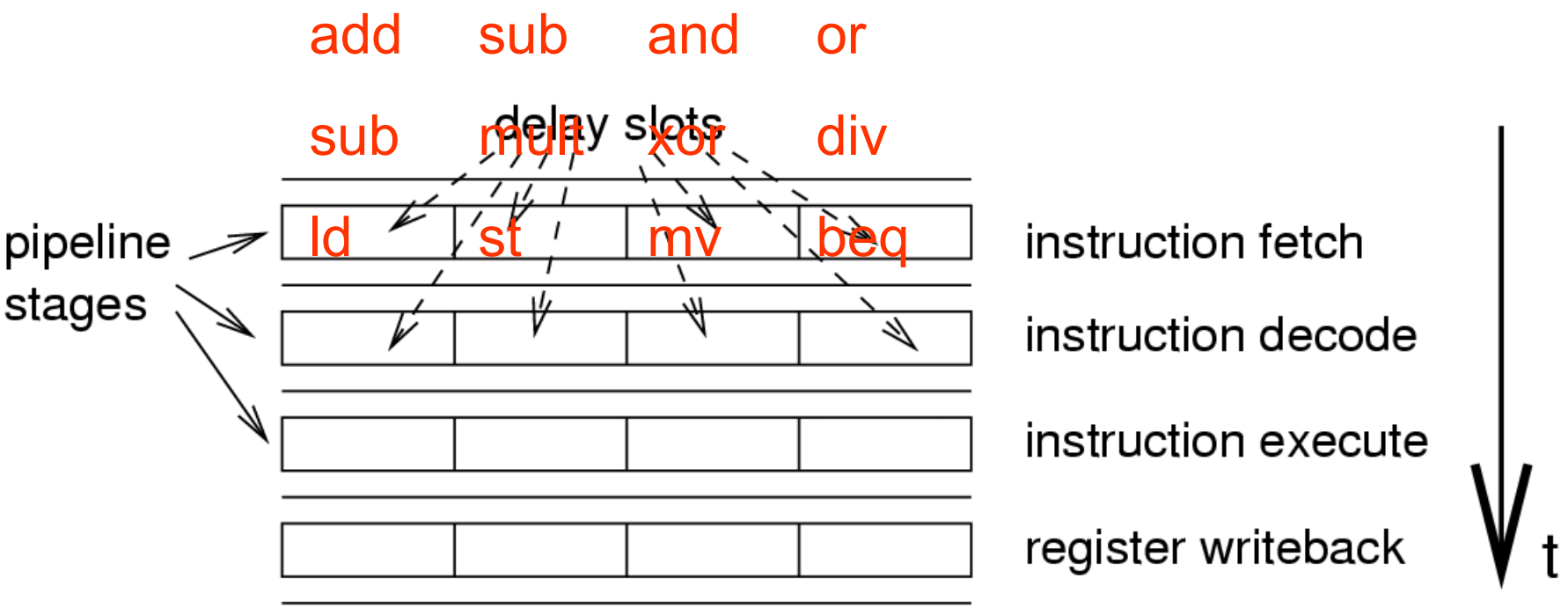
Itanium® 9300 (Tukwila), 2010



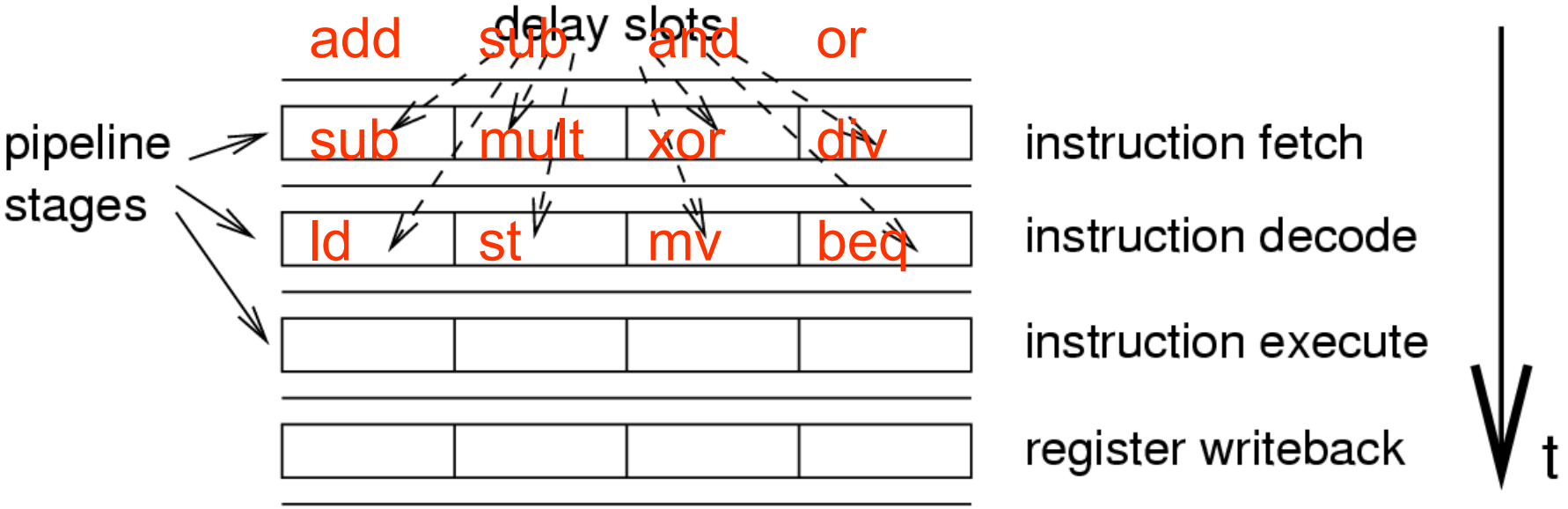
- 2 G transistors
- 4 cores
- 2-fold hyper-threading
- 8 threads
- 1.5 GHz at 1.3V

http://www.intel.com/cd/corporate/pressroom/emea/deu/442093.htm

Large # of delay slots, a problem of VLIW processors

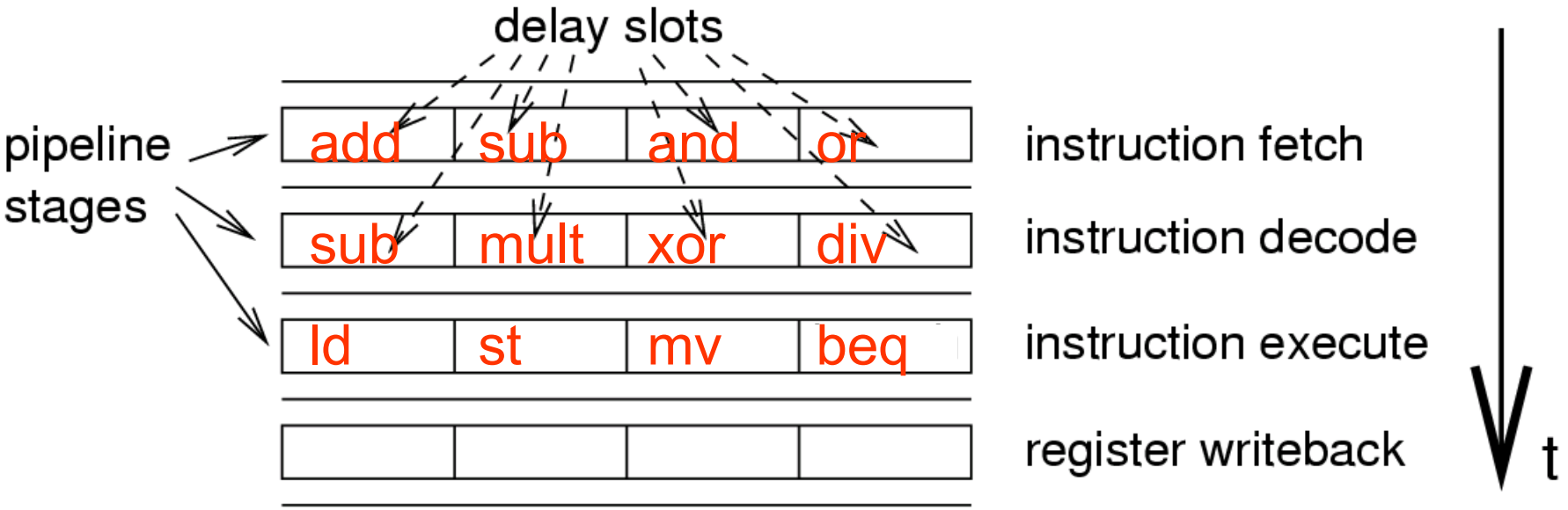


Large # of delay slots, a problem of VLIW processors



Large # of delay slots, a problem of VLIW processors

The execution of many instructions has been started before it is realized that a branch was required.



Nullifying those instructions would waste compute power

- ☞ Executing those instructions is declared a feature, not a bug.
- ☞ How to fill all “delay slots“ with useful instructions?
- ☞ Avoid branches wherever possible.

Predicated execution: Implementing IF-statements “branch-free“

Conditional Instruction “[c] I“ consists of:

- condition c (some expression involving condition code regs)
- instruction I

c = true ☞ I executed
c = false ☞ NOP

Predicated execution: Implementing IF-statements “branch-free“: TI C6xxx

```
if (c)
{ a = x + y;
  b = x + z;
}
else
{ a = x - y;
  b = x - z;
}
```

Conditional branch

```
      [c] B L1
          NOP 5
          B L2
          NOP 4
          SUB x,y,a
       ||  SUB x,z,b
L1:       ADD x,y,a
       ||  ADD x,z,b
L2:
```

max. 12 cycles

Predicated execution

```
   [c]  ADD x,y,a
|| [c]  ADD x,z,b
|| [!c] SUB  x,y,a
|| [!c] SUB  x,z,b
```

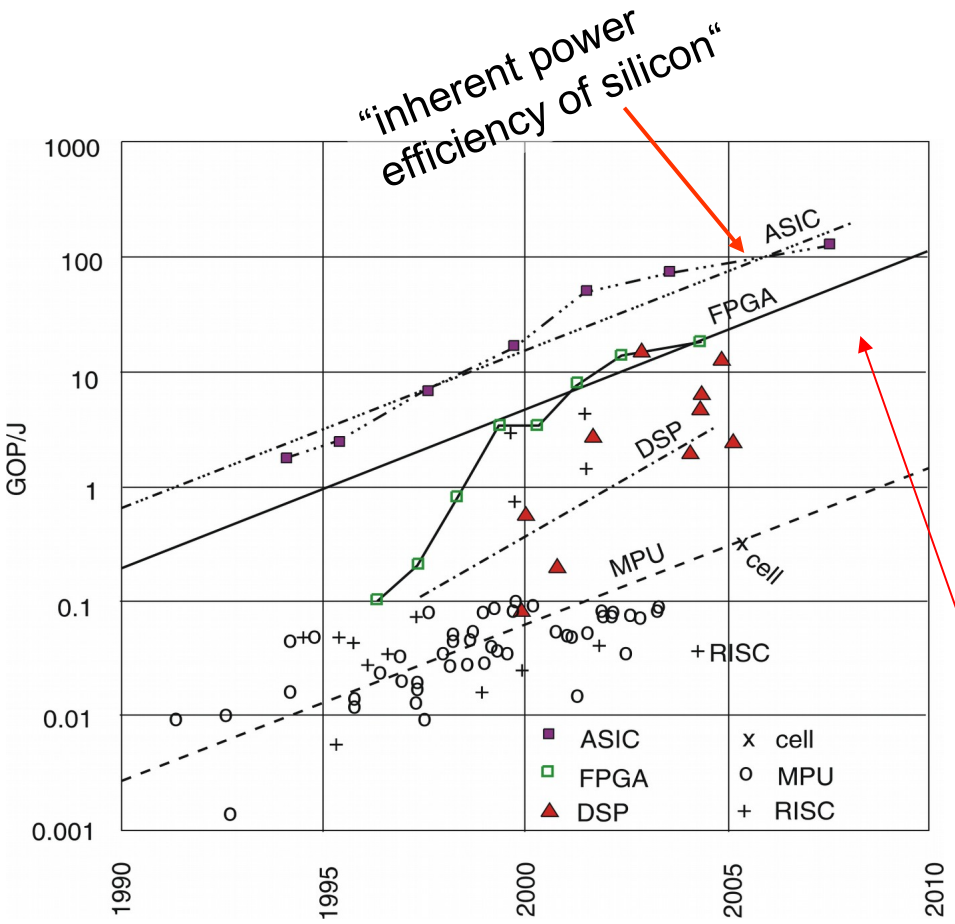
1 cycle

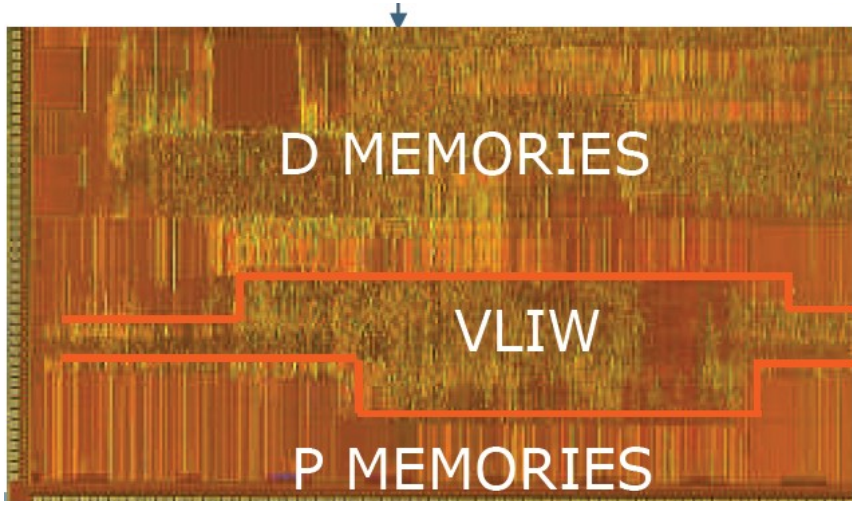
Microcontrollers
- MHS 80C51 as an example -

- 8-bit CPU optimised for control applications
- Extensive Boolean processing capabilities
- 64 k Program Memory address space
- 64 k Data Memory address space
- 4 k bytes of on chip Program Memory
- 128 bytes of on chip data RAM
- 32 bi-directional and individually addressable I/O lines
- Two 16-bit timers/counters
- Full duplex UART
- 6 sources/5-vector interrupt structure with 2 priority levels
- On chip clock oscillators
- Very popular CPU with many different variations

Features 4 Embedded Systems

Energy efficiency reached with VLIW processors

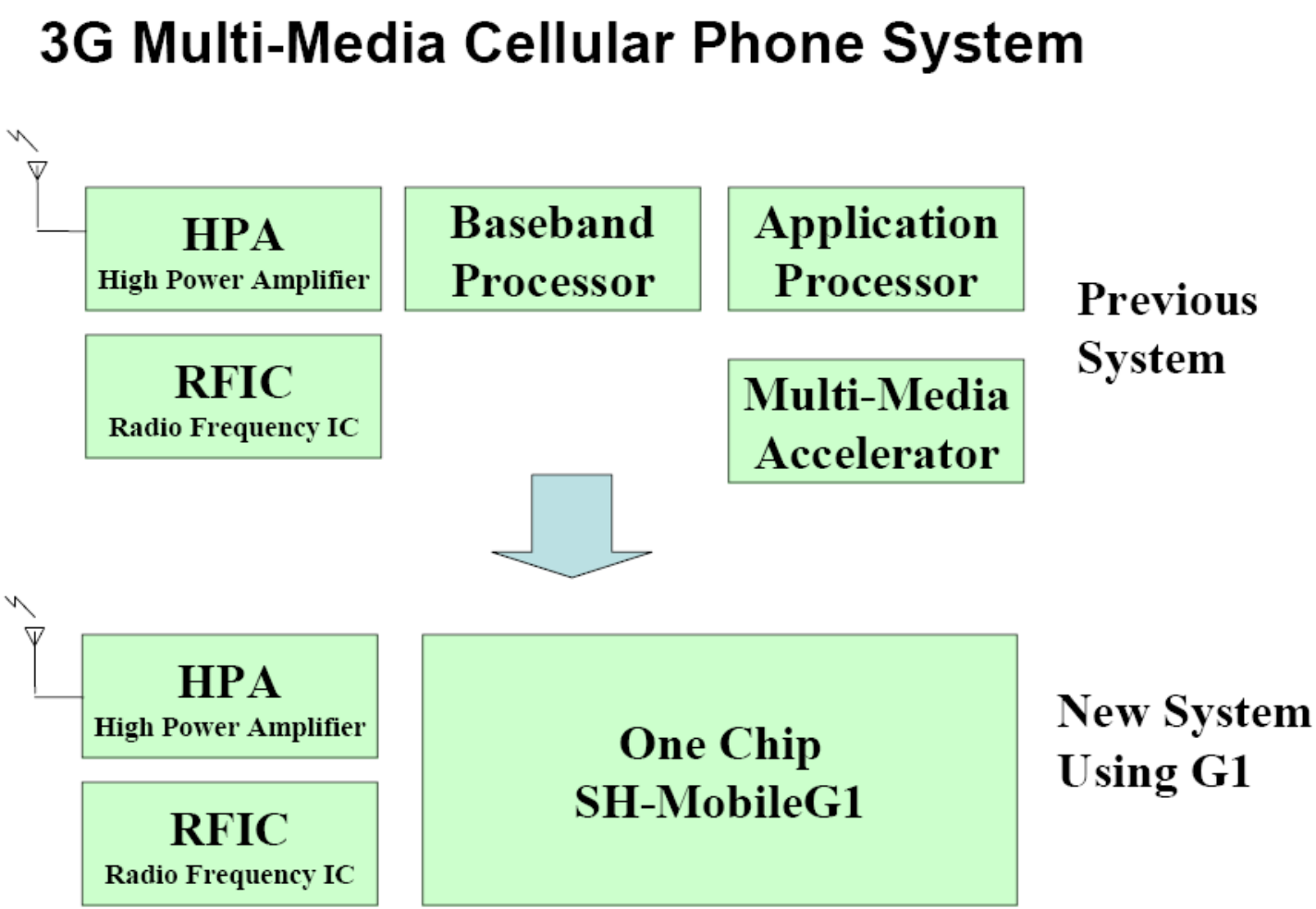




© Silicon Hive
41 Issue VLIW for SDR
130 nm, 1,2 V, 6,5mm², 16 bit
30 operations/cycle (OFDM)
150 MHz, 190 mW (incl. SRAMs)
24 GOPs/W, ~1/5 IPE

© Hugo De Man: From the Heaven of Software to the Hell of Nanoscale Physics: An Industry in Transition, Keynote Slides, ACACES, 2007

Trend: multiprocessor systems-on-a-chip (MPSoCs)



http://www.mpsoc-forum.org/2007/slides/Hattori.pdf

technische universität
dortmund

fakultät für informatik
informatik 12

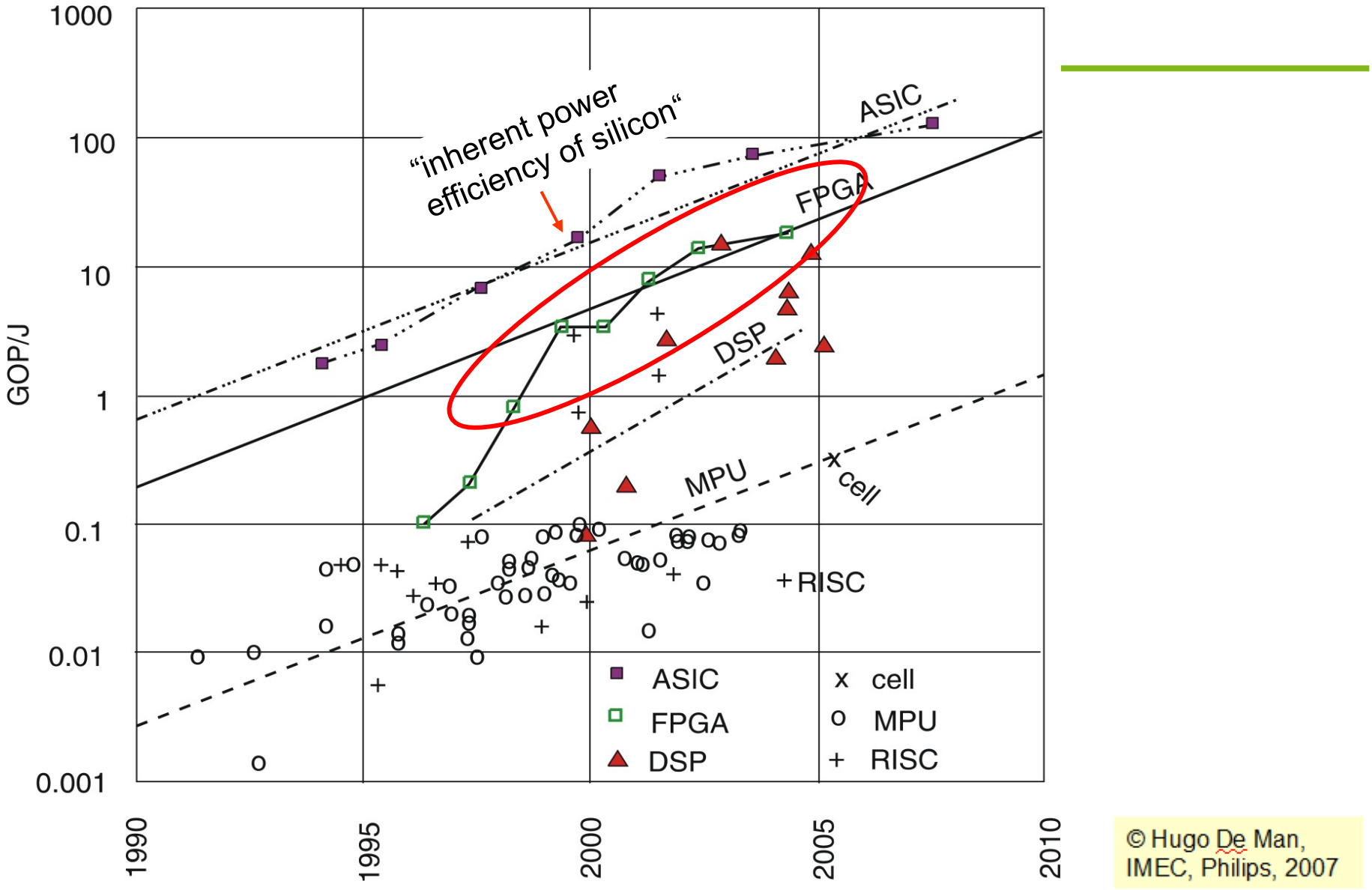
Embedded System Hardware

- Reconfigurable Hardware -

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 11 月 20 日

Energy Efficiency of FPGAs



© Hugo De Man, IMEC, Philips, 2007

Reconfigurable Logic

Custom HW may be too expensive, SW too slow.

Combine the speed of HW with the flexibility of SW

- ☞ HW with programmable functions and interconnect.
- ☞ Use of configurable hardware;
  common form: field programmable gate arrays (FPGAs)

Applications:

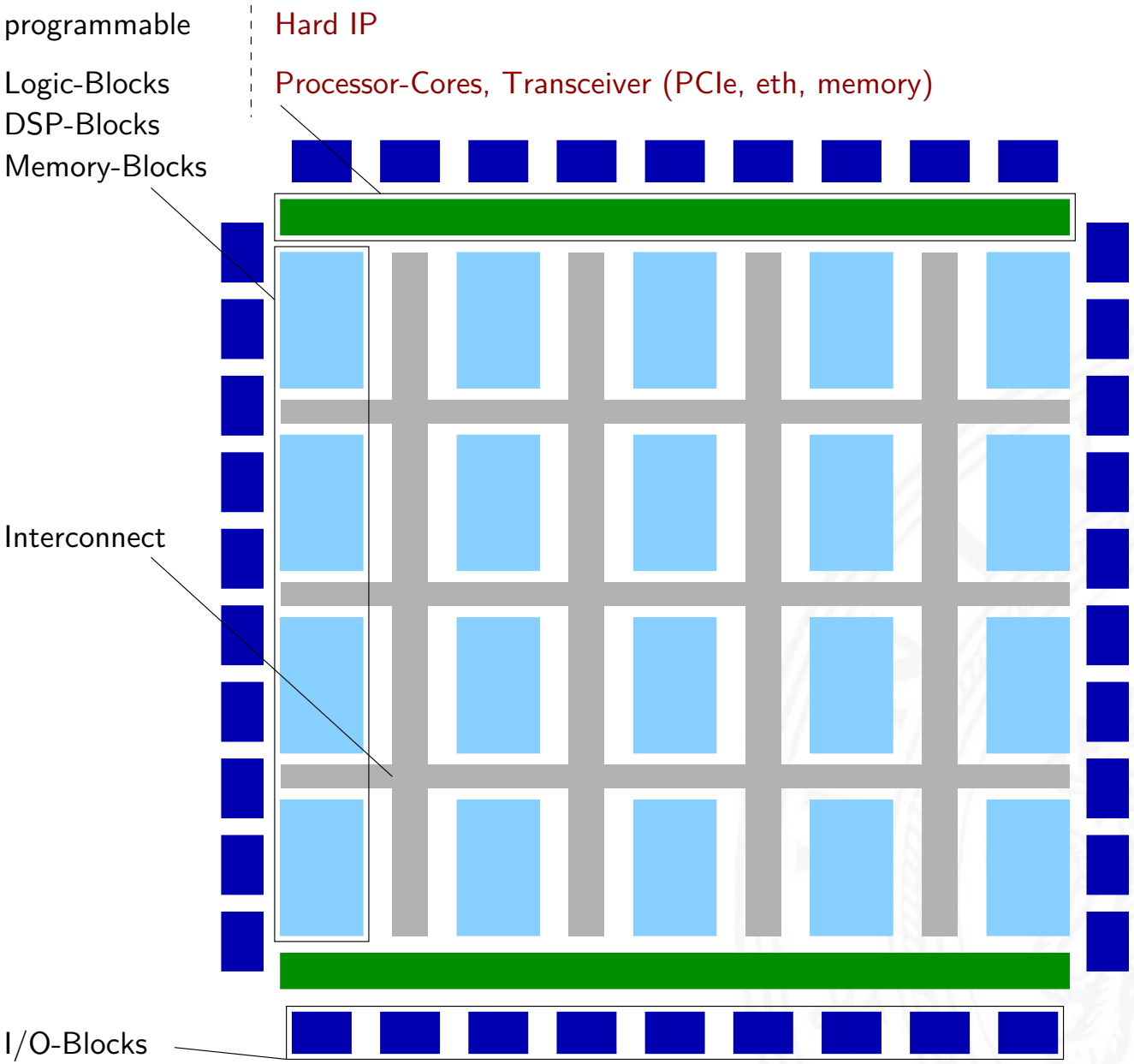
- algorithms like de/encryption,
- pattern matching in bioinformatics,
- high speed event filtering (high energy physics),
- high speed special purpose hardware.

Very popular devices from

- XILINX, Intel (Altera), Actel and others

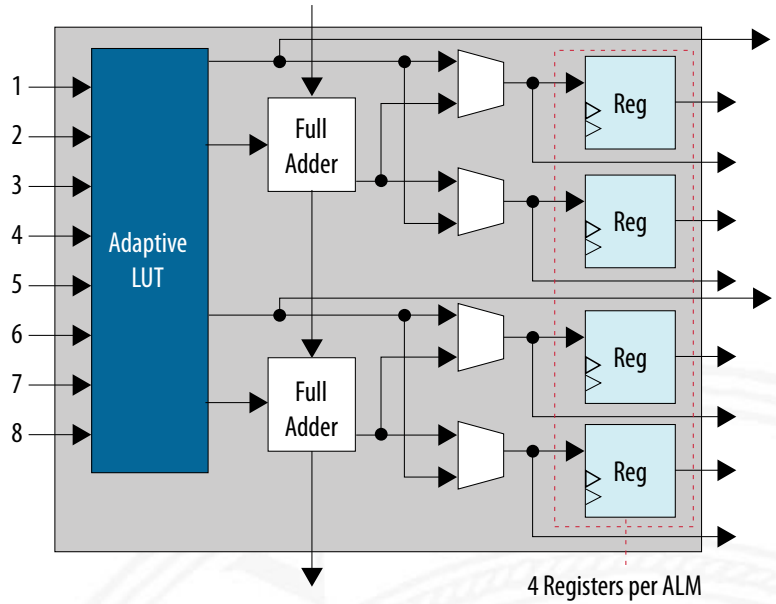
– Beginn –

FPGA: programmierbare + fest vorgebene Bereiche

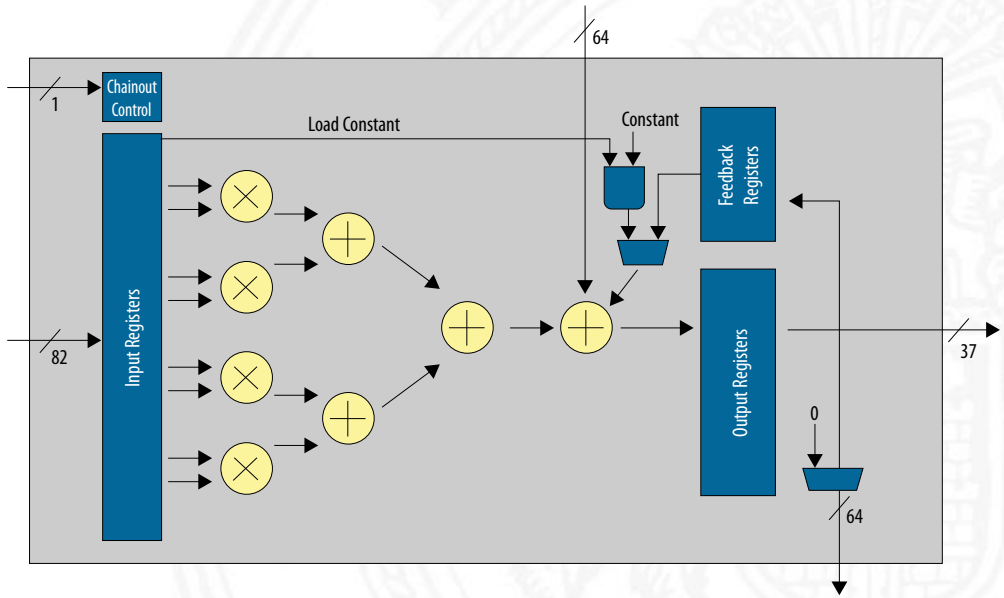


Beispiel

- Logic-Block



- DSP-Block

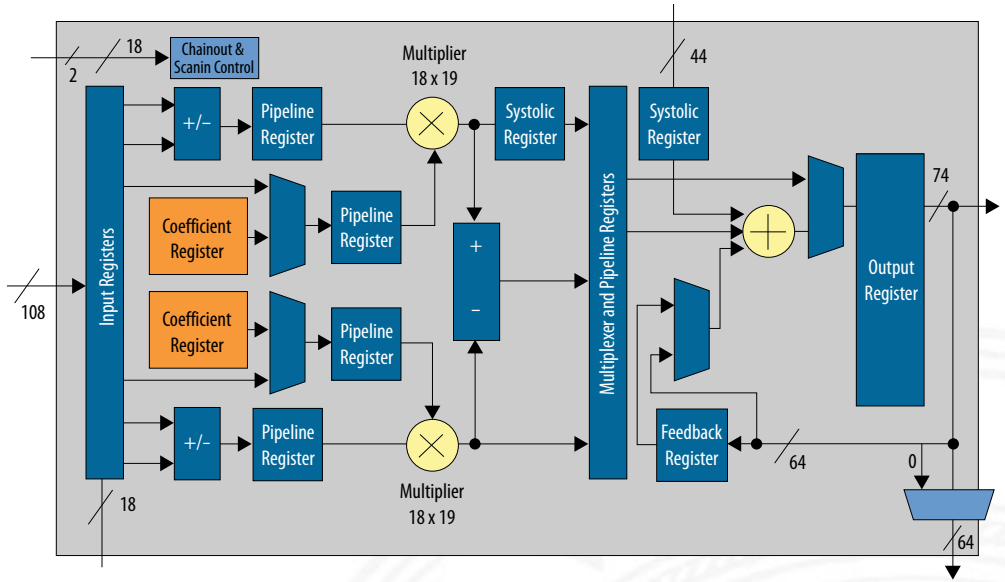


Intel Agilex FPGA Advanced Information Brief [Intel-Agilex]

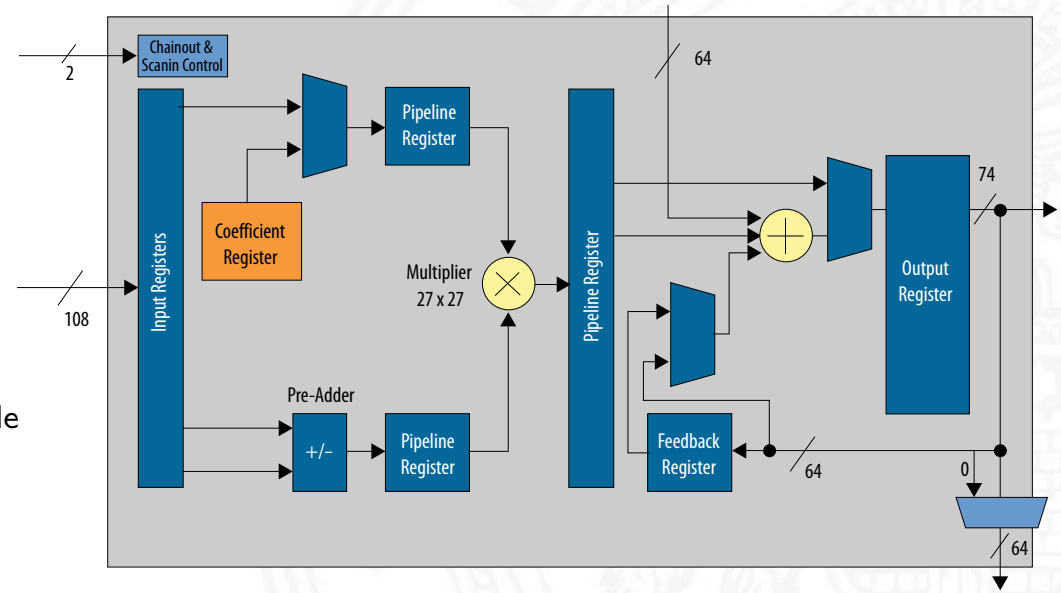
Beispiel (cont.)

- DSP-Block

Standard Precision Fixed Point Mode

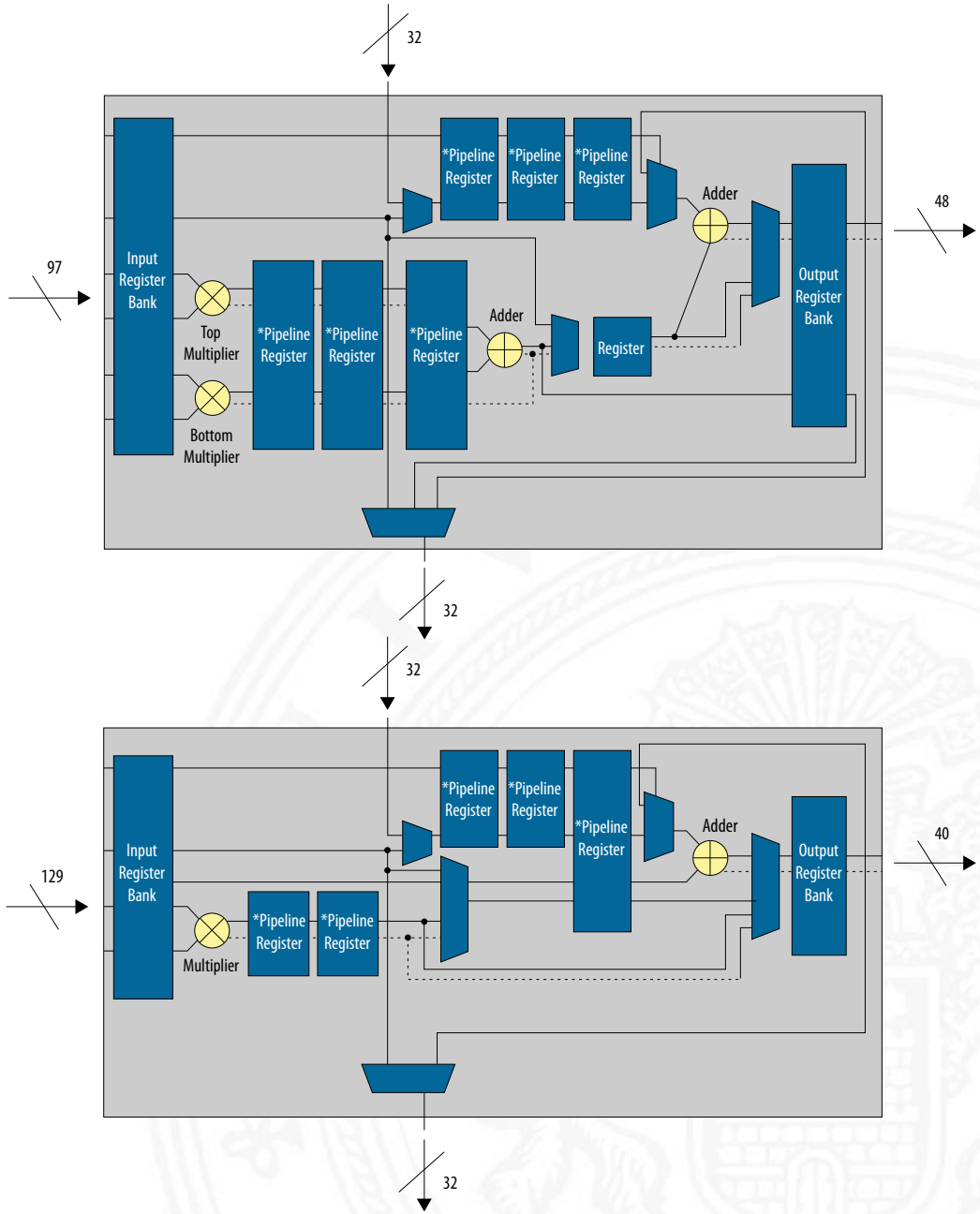


High Precision Fixed Point Mode



Beispiel (cont.)

- DSP-Block



Half Precision Floating Point (16-bit)

Single Precision Floating Point (32-bit)

- Intel Agilex [IntelFPGA]

Intel Agilex F-Series Device Names	Logic Elements (LE)	eSRAM Blocks	eSRAM Mbits	M20K Blocks	M20K Mbits	MLAB Counts	MLAB Mbits	Variable Precision DSP Blocks	18x19 Multipliers
AGF 004	392,000	0	0	1,900	38	6644	4.3	1,640	2.3K
AGF 006	573,480	0	0	2,844	56	9720	6.2	1,640	3.3K
AGF 008	764,640	0	0	3,792	74	12960	8.3	2,296	4.6K
AGF 012	1,200,000	2	36	5,568	110	20338	13	4,000	8K
AGF 014	1,437,240	2	36	7,110	139	24,360	15.6	4,510	9K
AGF 022	2,200,000	0	0	11,616	210	37288	21	6,250	12.5K
AGF 027	2,692,760	0	0	13,272	259	45,640	29.2	8,736	17K

FPGA Komplexität (cont.)

- Xilinx Versal [Xilinx]

	VC1352	VC1502	VC1702	VC1802	VC1902
AI Engines	128	217	310	300	400
AI Engine Data Memory Blocks	1,024	1,736	2,480	2,400	3,200
AI Engine Data Memory (Mb)	32	54.25	77.5	75	100
DSP Engines	928	1,312	1,272	1,600	1,968
System Logic Cells	539,840	797,440	1,020,880	1,585,938	1,968,400
CLB Flip-Flops	493,568	729,088	933,376	1,450,000	1,799,680
LUTs	246,784	364,544	466,688	725,000	899,840
Distributed RAM (Mb)	7.5	11.1	14.2	22.1	27.5
Block RAM Blocks	499	547	826	800	967
Block RAM (Mb)	17.5	19.2	29.0	28.1	34.0
UltraRAM Blocks	151	215	402	325	463
UltraRAM (Mb)	42.5	60.5	113.1	91.4	130.2
Accelerator RAM Blocks	1	0	1	0	0
Accelerator RAM (Mb)	32	0	32	0	0
APU	Dual-core Arm Cortex-A72; 48KB/32KB L1 Cache w/ parity and ECC; 1MB L2 Cache w/ ECC				
RPU	Dual-core Arm Cortex-R5; 32KB/32KB L1 Cache, and TCM w/ECC				
Memory	256KB On-Chip Memory w/ECC				
Connectivity	Ethernet (x2); UART (x2); CAN-FD (x2); USB 2.0 (x1); SPI (x2); I2C (x2)				
NoC Master / Slave Ports	10	14	18	28	28
DDR Bus Width	128	128	128	256	256
DDR Memory Controllers	2	2	2	4	4
CCIX & PCIe (CPM)	–	1 x Gen4x16, CCIX	–	1 x Gen4x16, CCIX	1 x Gen4x16, CCIX
PCI Express	1 x Gen4x8	4 x Gen4x8	1 x Gen4x8	4 x Gen4x8	4 x Gen4x8
Multirate Ethernet MAC	1	4	3	4	4
XPIO	378	378	378	648	648
HDIO	44	44	44	44	44
GTY Transceivers (32.75Gb/s)	8	44	24	44	44

FPGA Komplexität (cont.)

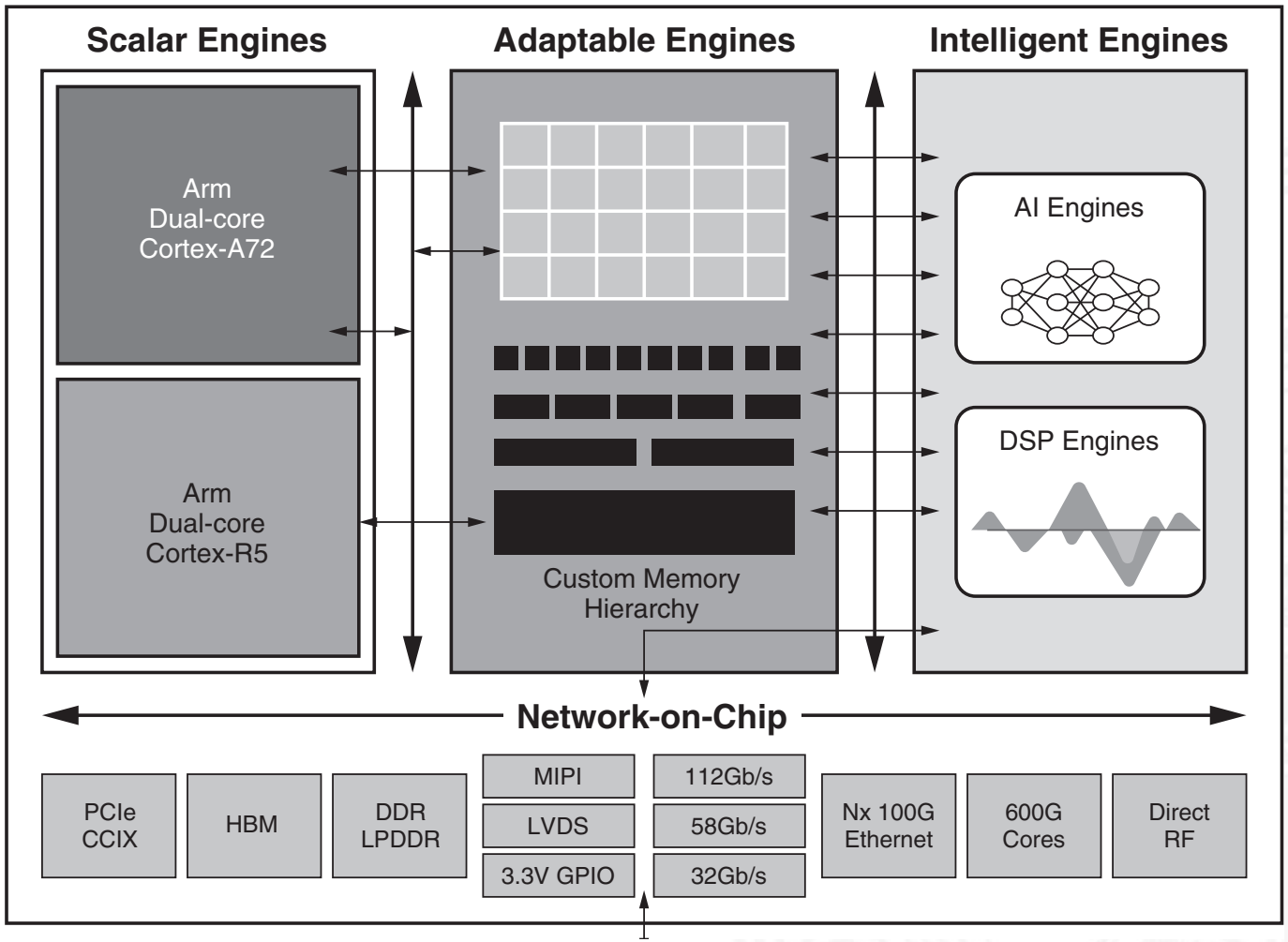
- Xilinx Versal [Xilinx]

	VM1102	VM1302	VM1402	VM1502	VM1802	VM2502	VM2602	VM2702	VM2902
DSP Engines	472	736	1,504	1,312	1,968	3,984	1,880	2,500	3,080
System Logic Cells	352,240	571,760	1,001,840	799,440	1,968,400	2,029,720	1,263,360	1,804,688	2,153,760
CLB Flip-Flops	322,048	522,752	915,968	729,088	1,799,680	1,855,744	1,155,072	1,650,000	1,969,152
LUTs	161,024	261,376	457,984	364,544	899,840	927,872	577,536	825,000	984,576
Distributed RAM (Mb)	4.9	8.0	14.0	10.7	27.5	28.3	17.6	25.2	30.0
Block RAM Blocks	216	452	1,124	547	967	1,365	1,552	2,100	2,572
Block RAM (Mb)	7.6	15.9	39.5	19.2	34.0	48.0	54.6	73.8	90.4
UltraRAM Blocks	96	168	168	215	463	701	424	600	724
UltraRAM (Mb)	27.0	47.3	47.3	60.5	130.2	197.2	119.3	168.8	203.6
APU	Dual-core Arm Cortex-A72; 48KB/32KB L1 Cache w/ parity and ECC; 1MB L2 Cache w/ ECC								
RPU	Dual-core Arm Cortex-R5; 32KB/32KB L1 Cache, and TCM w/ECC								
Memory	256KB On-Chip Memory w/ECC								
Connectivity	Ethernet (x2); UART (x2); CAN-FD (x2); USB 2.0 (x1); SPI (x2); I2C (x2)								
NoC Master / Slave Ports	5	16	16	14	28	28	16	26	26
DDR Bus Width	64	128	256	128	256	288	384	384	384
DDR Memory Controllers	1	2	4	2	4	5	6	6	6
CCIX & PCIe (CPM)	_	_	1 x Gen4x16, CCIX	_	1 x Gen4x16, CCIX	1 x Gen4x16, CCIX	1 x Gen4x16, CCIX	1 x Gen4x16, CCIX	1 x Gen4x16, CCIX
PCI Express	1 x Gen4x8	2 x Gen4x8	4 x Gen4x8	2 x Gen4x8	4 x Gen4x8	1 x Gen4x8	1 x Gen4x8	2 x Gen4x8	2 x Gen4x8
Multirate Ethernet MAC	1	2	2	4	4	1	2	2	2
XPIO	216	432	648	378	648	702	756	702	702
HDIO	22	44	44	44	44	44	22	44	44
GTY Transceivers (32.75Gb/s)	12	24	24	44	44	16	20	32	40
GTM Transceivers (58Gb/s)	0	0	0	0	0	28	32	44	52

⇒ FPGAs derzeit komplexeste ICs ($\approx$ 50 Milliarden Transistoren)

Zukunftskonzepte

- Rechenbeschleuniger in Workstations und PCs
- dynamische (Teil-) Konfiguration zur Programmlaufzeit



Xilinx Versal ACAP (Adaptive Compute Acceleration Platform) [Xilinx-Versal]

– Ende –

Memory

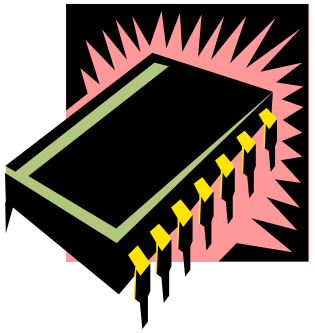
Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 11 月 20 日

Memory

Memories?

Oops!
Memories!



For the memory, efficiency is again a concern:

- capacity
- energy efficiency
- speed (latency and throughput); predictable timing
- size
- cost
- other attributes (volatile vs. persistent, etc)

Memory capacities expected to keep increasing

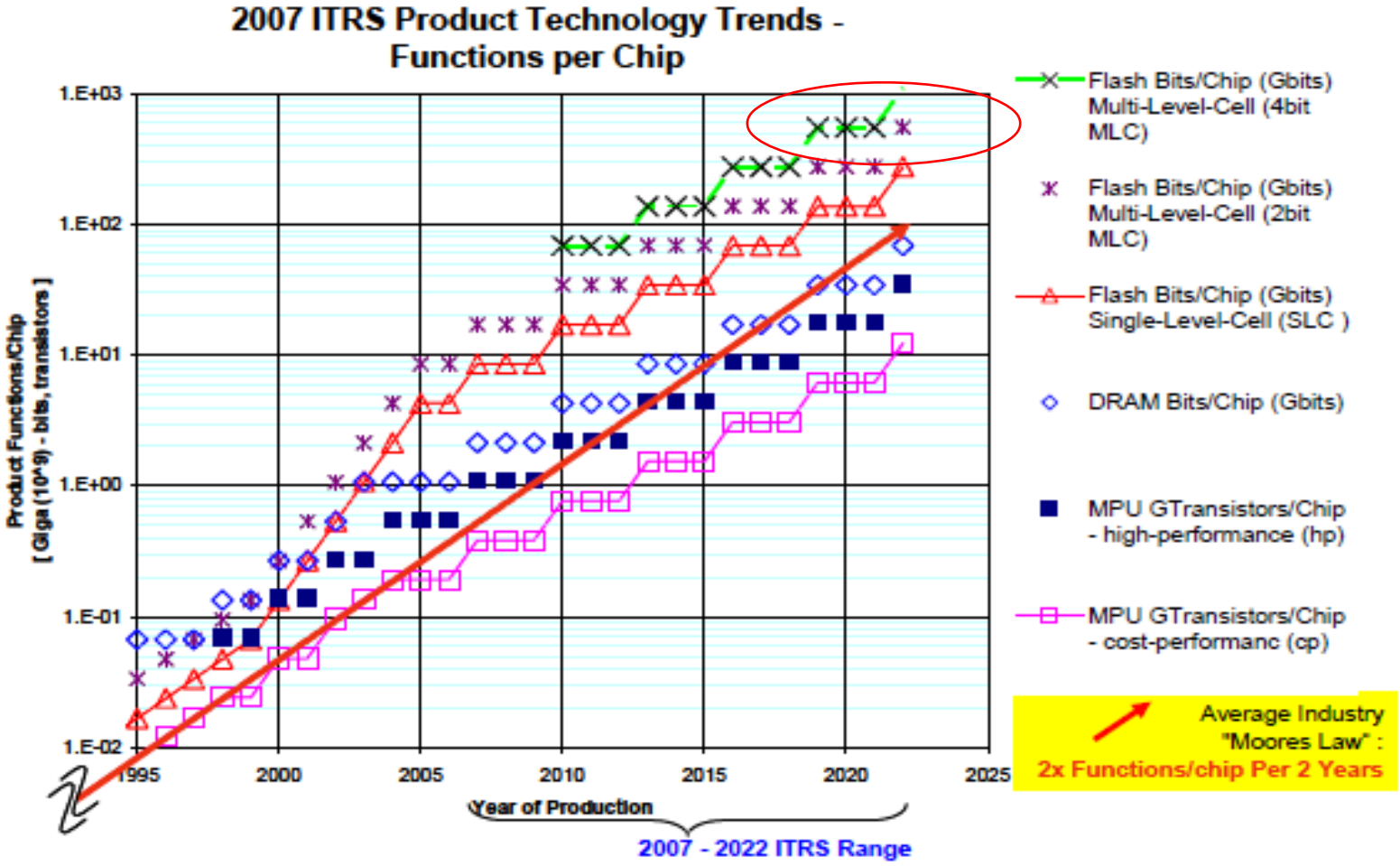
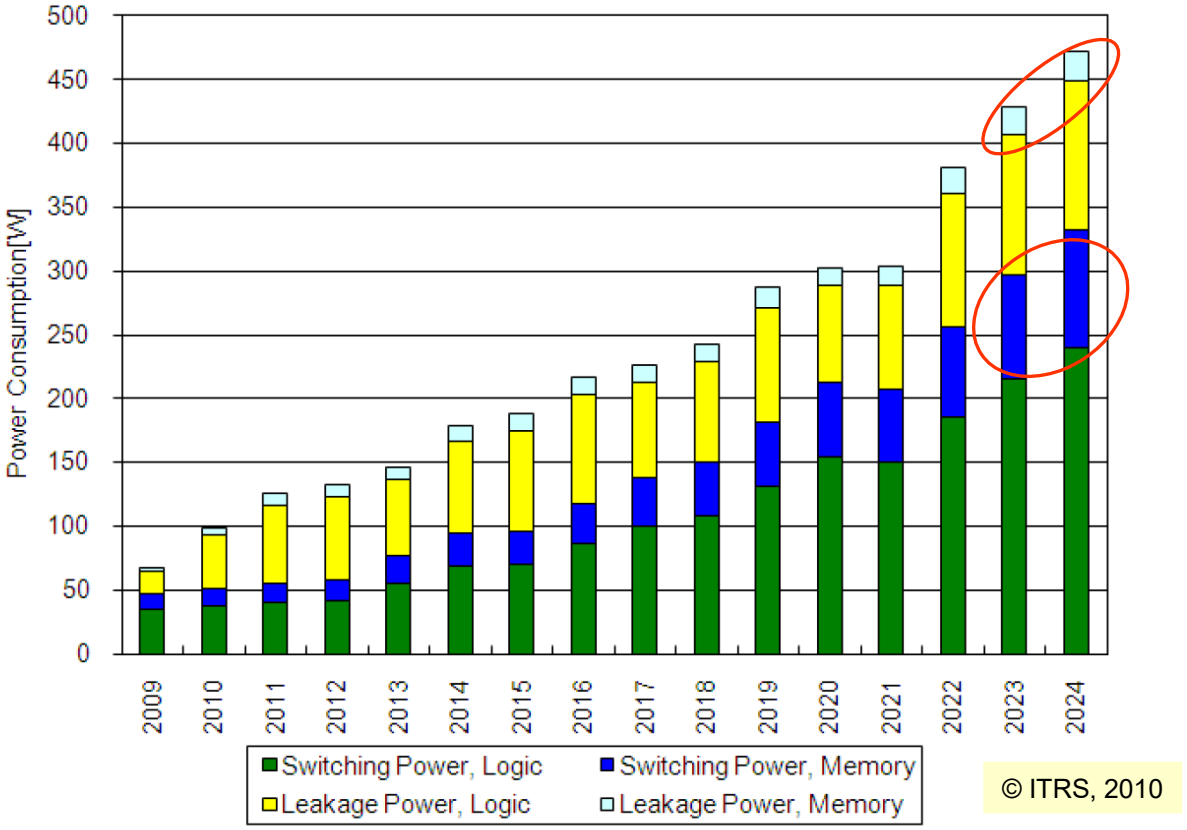


Figure ORTC2 ITRS Product Function Size Trends: MPU Logic Gate Size (4-transistor); Memory Cell Size [SRAM (6-transistor); Flash (SLC and MLC), and DRAM (transistor + capacitor)]--Updated

Where is the power consumed?
- Stationary systems -

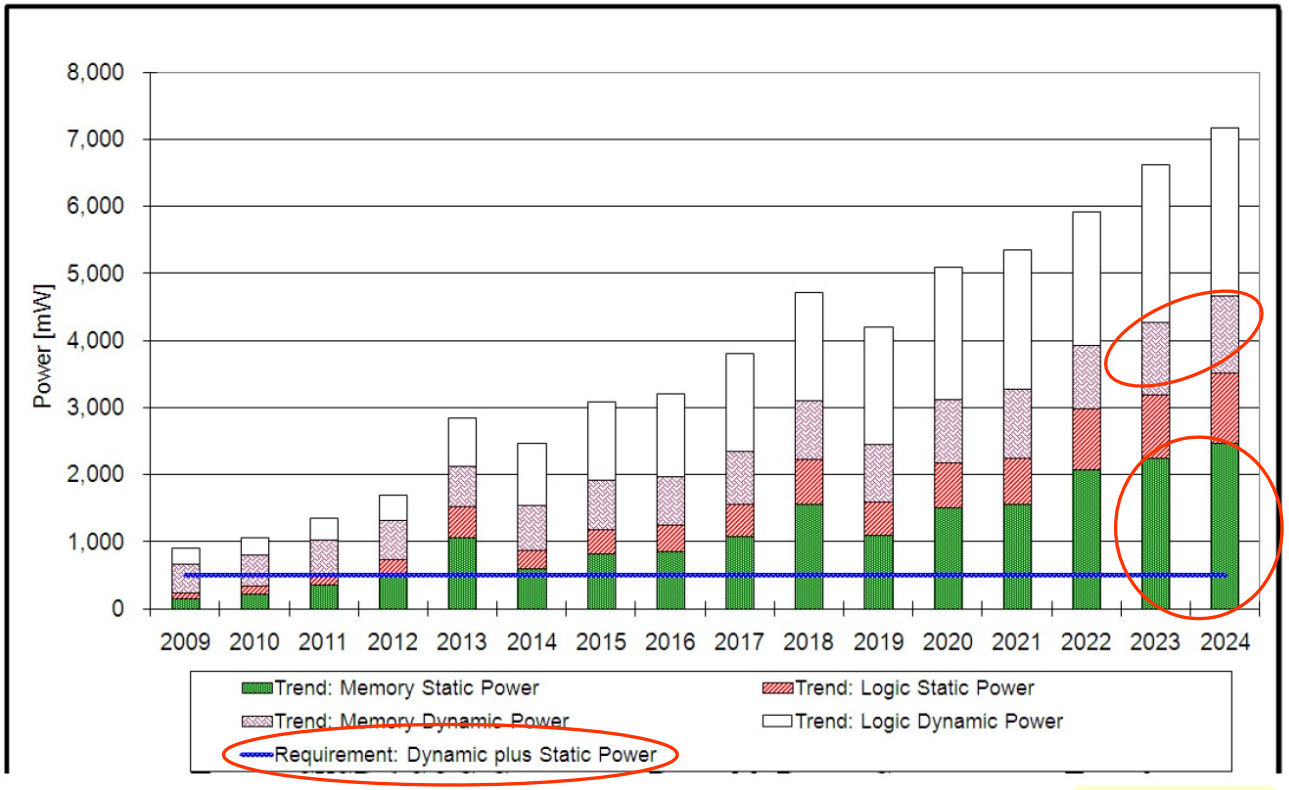
- According to *International Technology Roadmap for Semi-conductors* (ITRS), 2010 update, [www.itrs.net]



- Switching power, logic dominating
- Overall power consumption a nightmare for environmentalists

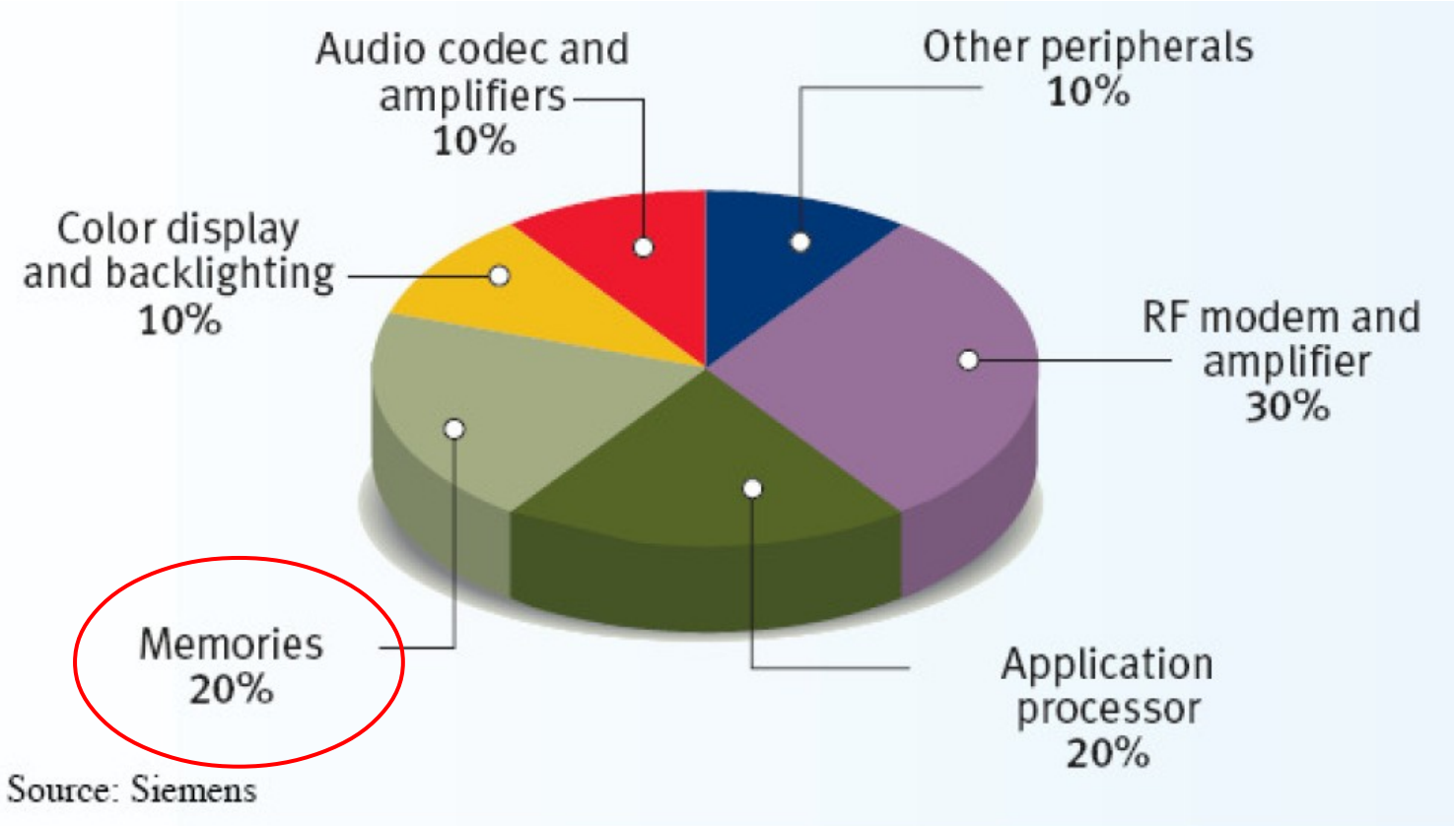
Where is the power consumed?
- Consumer portable systems -

- According to *International Technology Roadmap for Semi-conductors* (ITRS), 2010 update, [www.itrs.net]
- Based on current trends



- Memory and logic, static and dynamic relevant
- Following current trends will violate maximum power constraint (0.5-1 W).

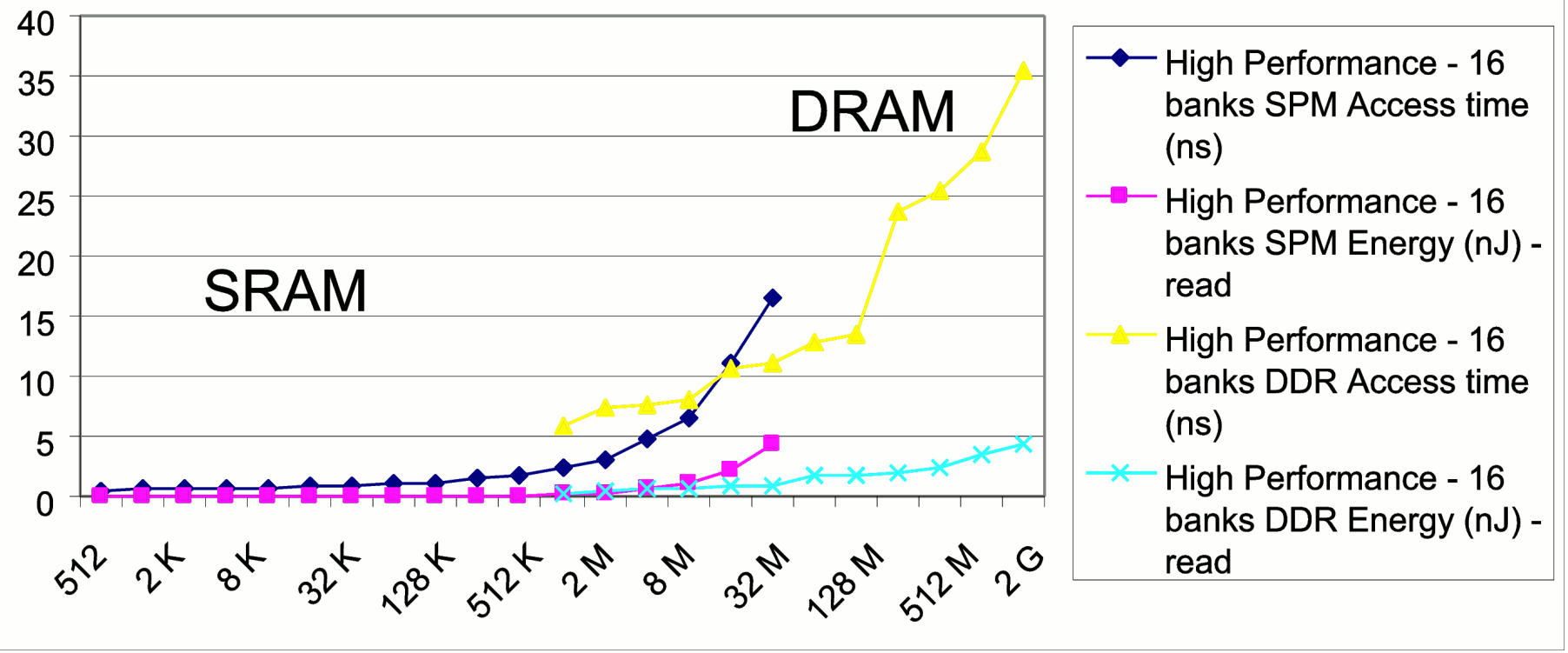
Memory energy significant even if we take display and RF of mobile device into account



O. Vargas (Infineon Technologies): Minimum power consumption in mobile-phone memory subsystems; Pennwell Portable Design - September 2005; Thanks to Thorsten Koch (Nokia/ Univ. Dortmund) for providing this source.

Energy consumption and access times of memories

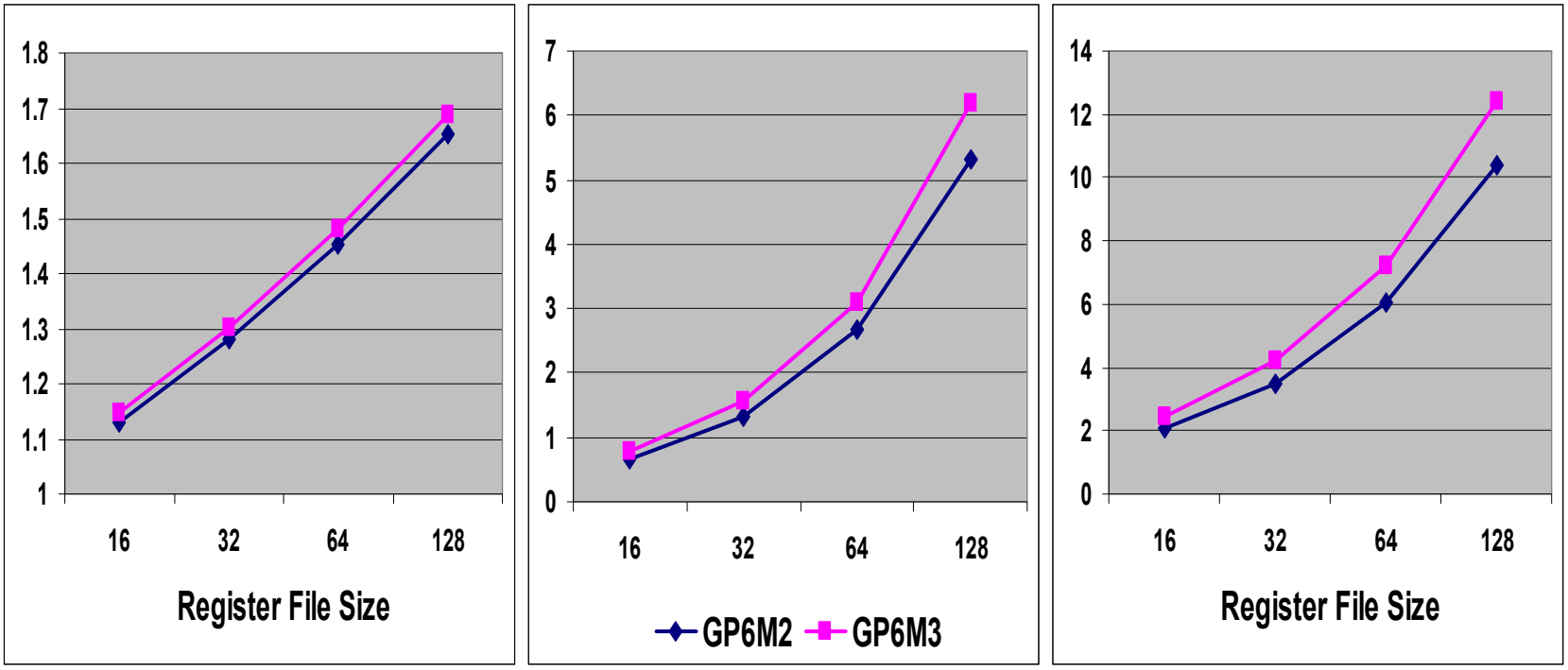
Example CACTI: Scratchpad (SRAM) vs. DRAM (DDR2):



16 bit read; size in bytes;
65 nm for SRAM, 80 nm for DRAM

Source: Olivera Jovanovic, TU Dortmund, 2011

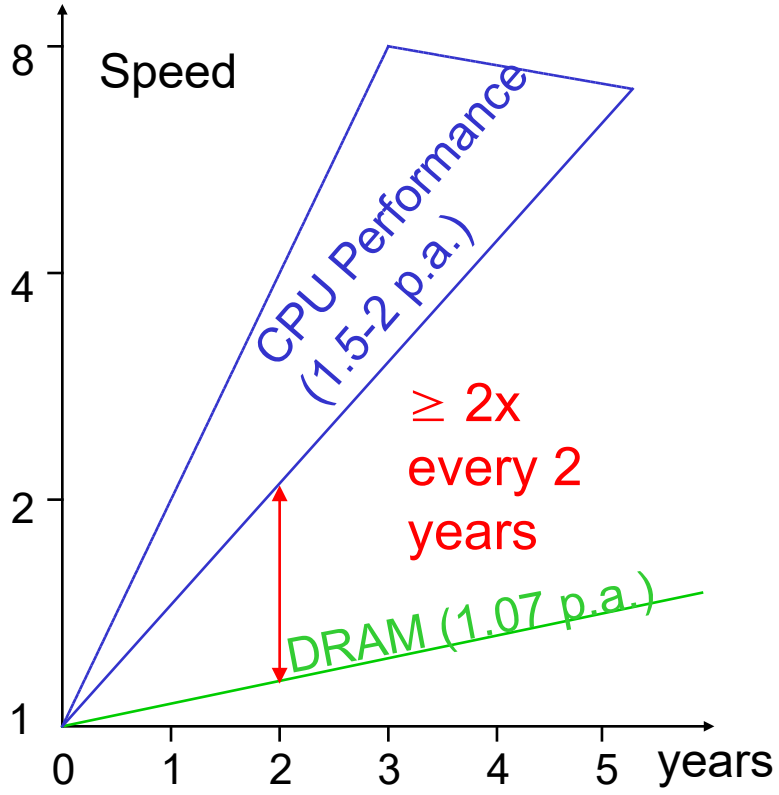
Access times and energy consumption for multi-ported register files



Source and © H. Valero, 2001

Trends for the Speeds

Speed gap between processor and main DRAM increases

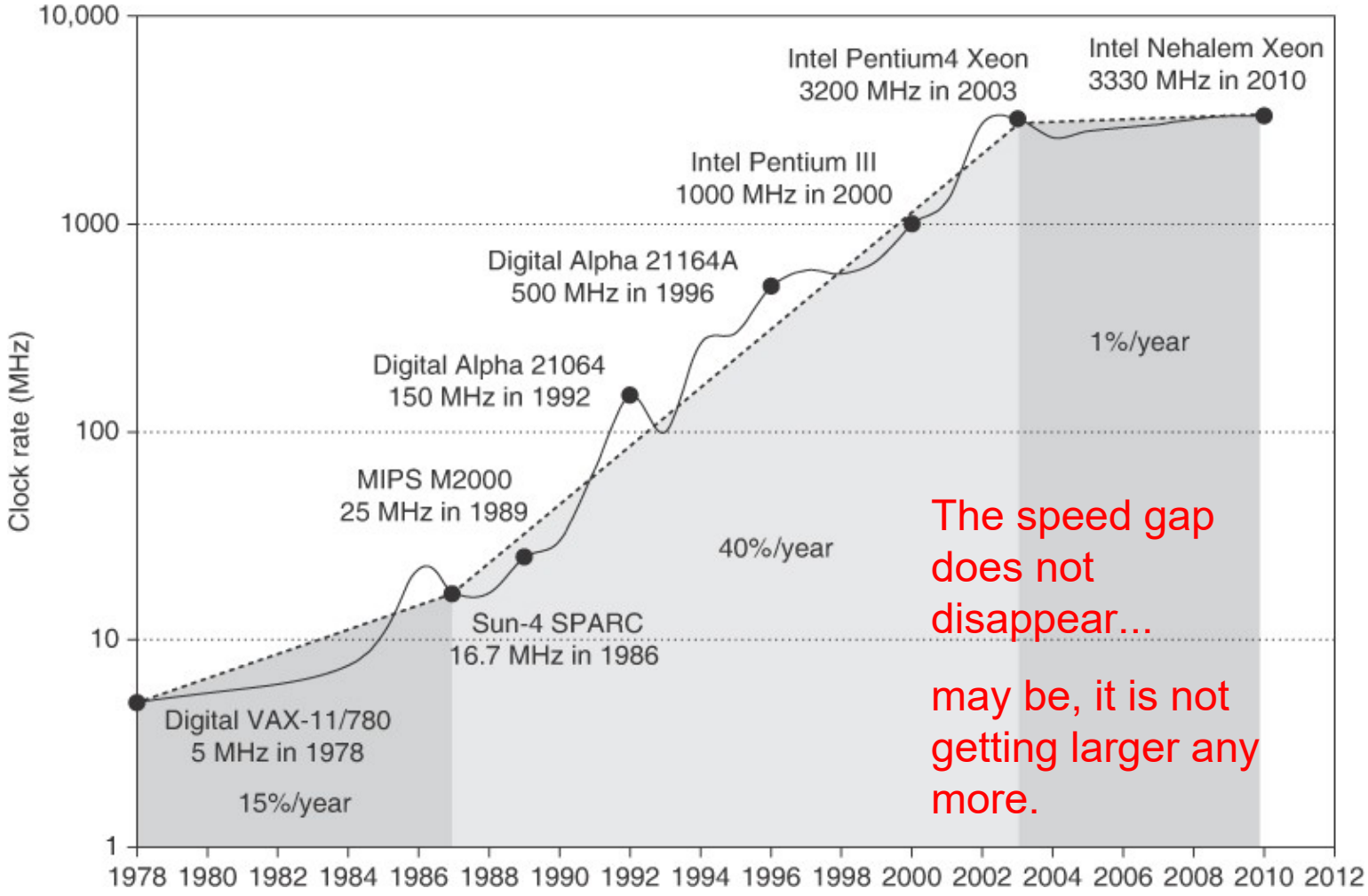


Similar problems also for embedded systems & MPSoCs

- ☞ Memory access times >> processor cycle times
- ☞ "Memory wall" problem

P. Machanik: Approaches to Addressing the Memory Wall, TR Nov. 2002, U. Brisbane

However, clock speed increases have come to a halt

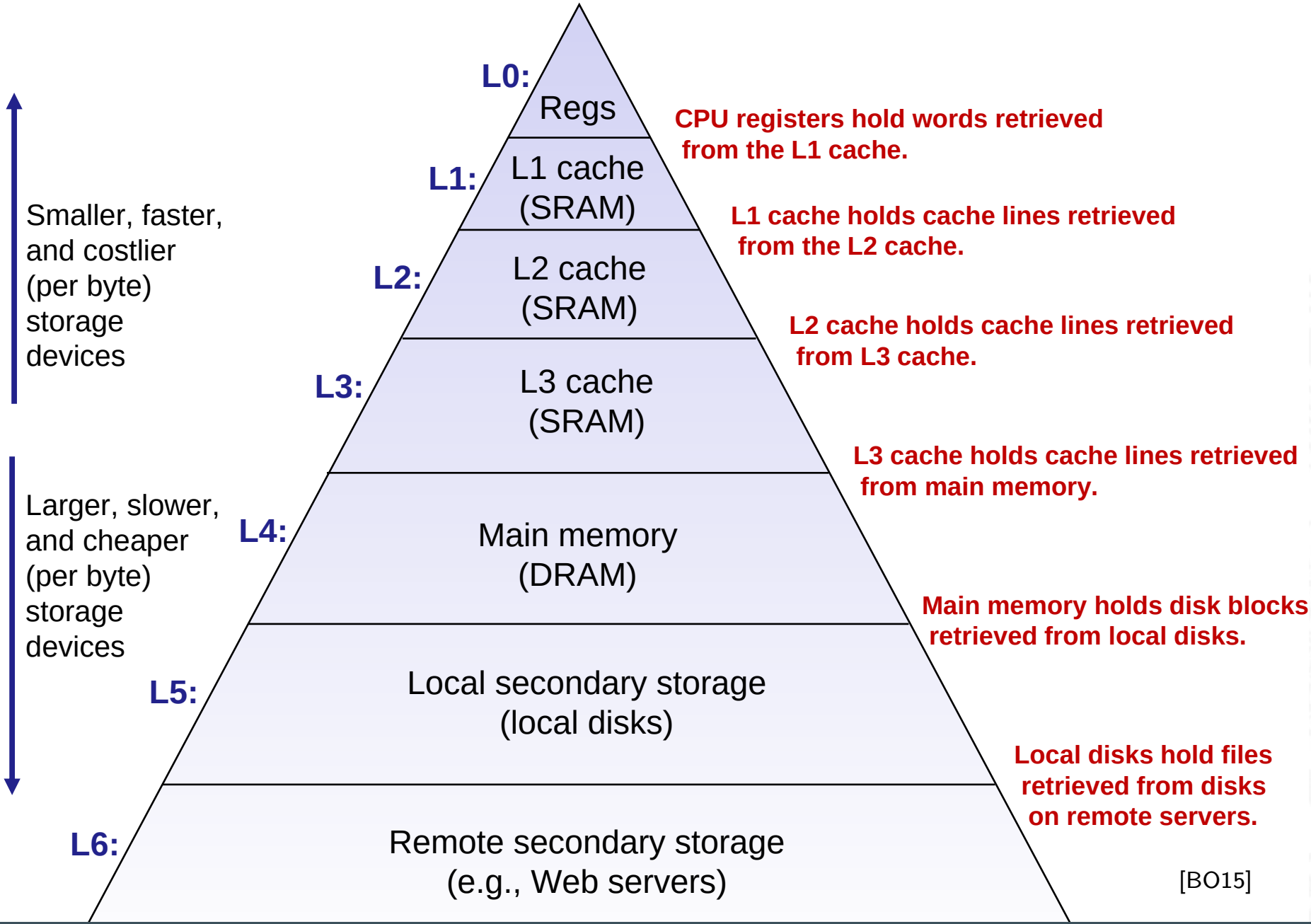


Hennessy/Patterson: Computer Architecture, 5th ed., 2011

Einschub: Speicherhierarchie / Caches

– Beginn –

Speicherhierarchie: Konzept



[BO15]

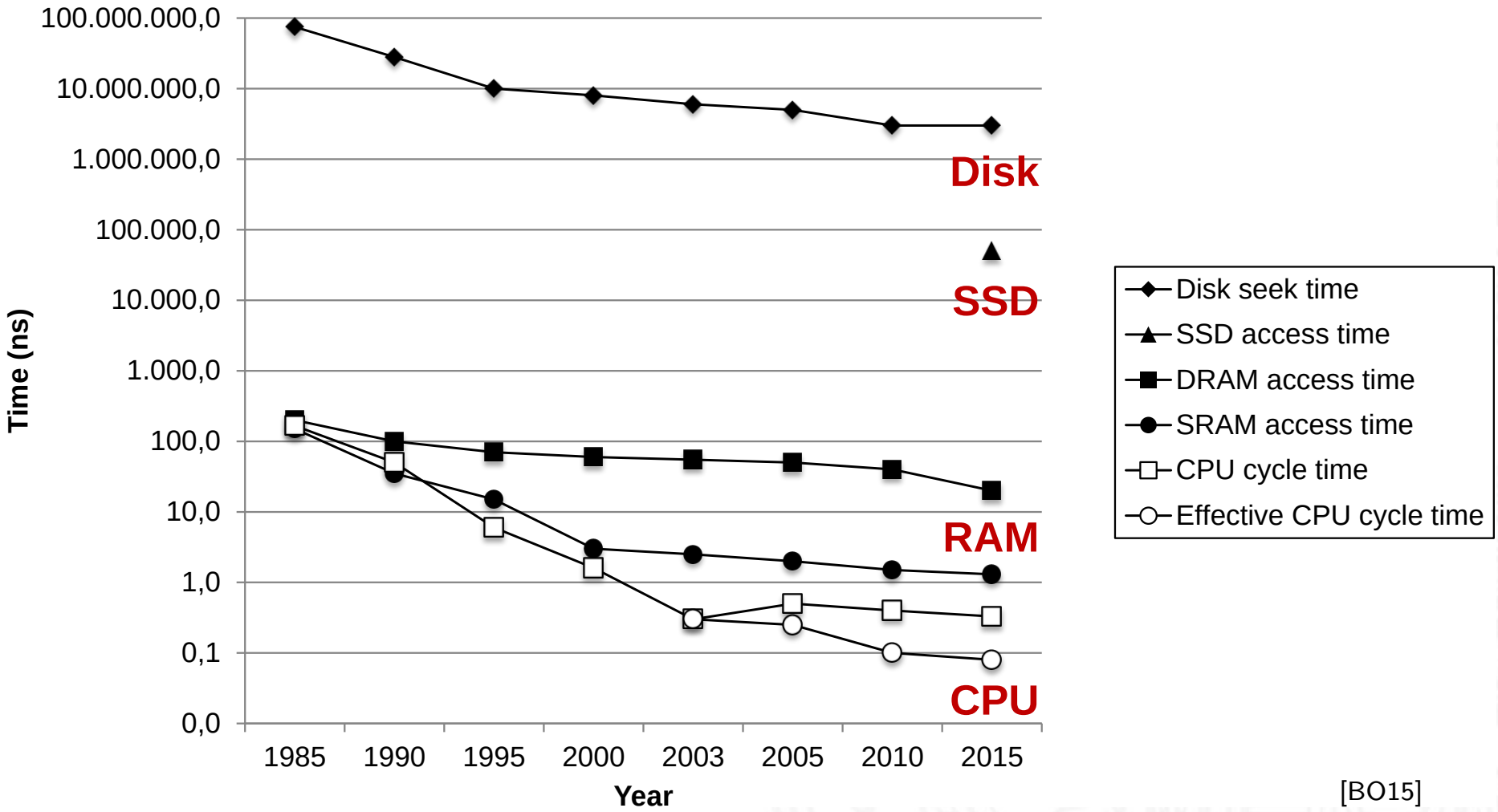
Eigenschaften der Speichertypen

Speicher	Vorteile	Nachteile
Register	sehr schnell	sehr teuer
SRAM	schnell	teuer, große Chips
DRAM	hohe Integration	Refresh nötig, langsam
Platten	billig, Kapazität	sehr langsam, mechanisch

Beispiel	Hauptspeicher	Festplatte	SSD
Latenz	8 ns	4 ms	0,2/0,4 ms
Bandbreite	25,6 GB/sec (pro Kanal, bis 4)	1,5 GB/sec	3/2 GB/sec (r/w)
Kosten/GB	6 €	2,5 ct. 1 TB: 25 €	25 ct.

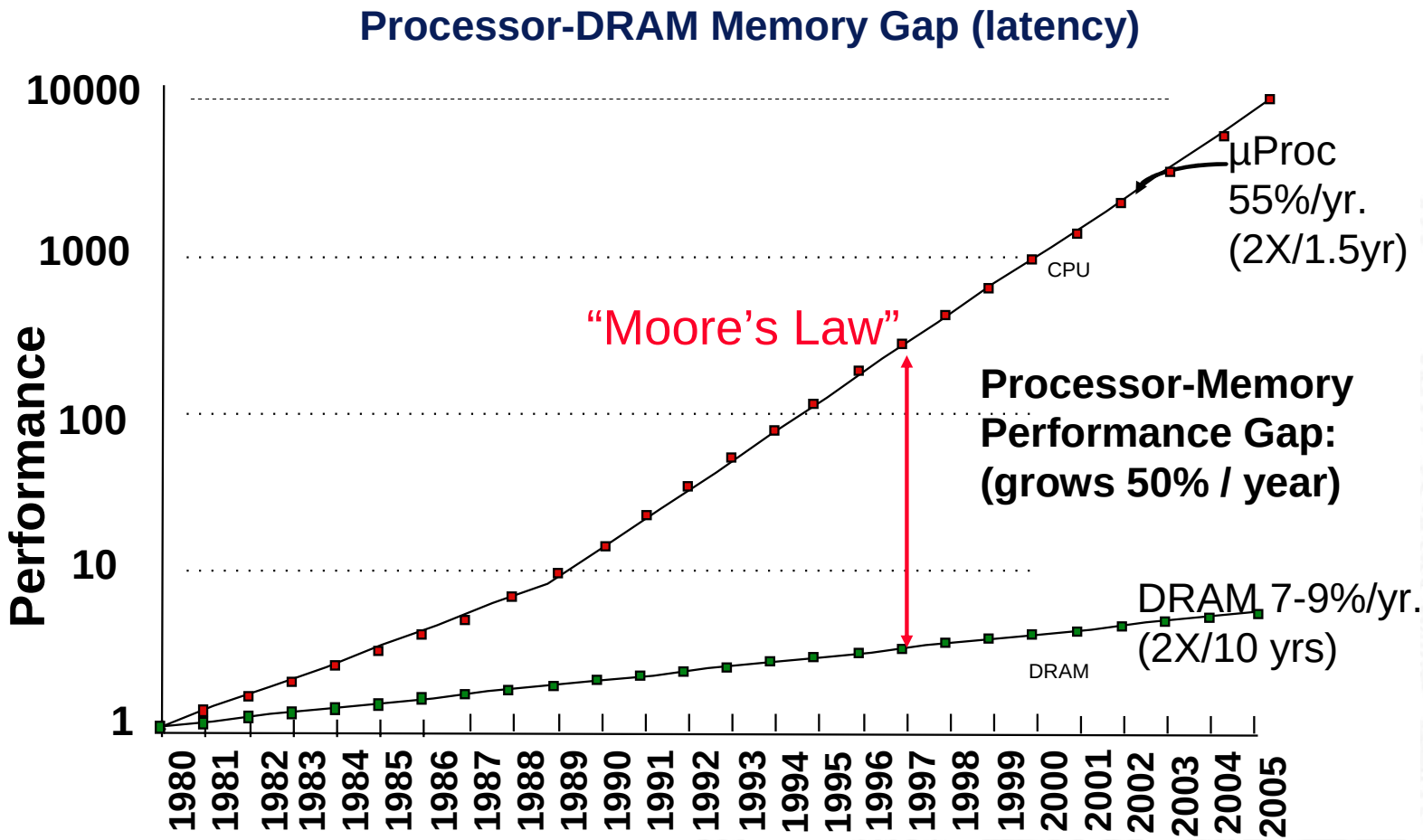
Prozessor-Memory Gap

- stetig wachsende Lücke zwischen CPU-, Memory- und Disk-Geschwindigkeiten



[BO15]

- „*Memory Wall*“: DRAM zu langsam für CPU



[PH16b]

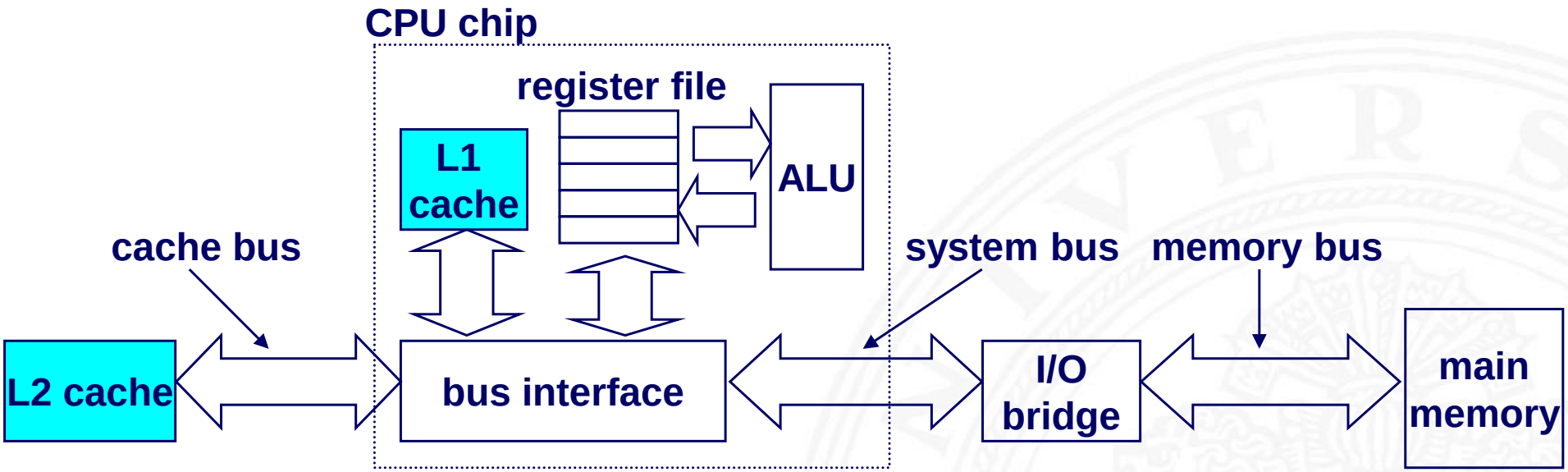
⇒ Cache als schneller Zwischenspeicher zum Hauptspeicher

Cache (cont.)

- technische Realisierung: SRAM
- transparenter Speicher
  - Cache ist für den Programmierer nicht sichtbar!
  - wird durch Hardware verwaltet
- ggf. getrennte Caches für Befehle und Daten
- enthält Hauptspeicherblöcke mit erhöhter Zugriffswahrscheinlichkeit
- basiert auf Prinzip der Lokalität von Speicherzugriffen durch ein laufendes Programm
  - ca. 80% der Zugriffe greifen auf 20% der Adressen zu
  - manchmal auch 90% / 10% oder noch besser
- `https://de.wikipedia.org/wiki/Cache`
  `https://en.wikipedia.org/wiki/CPU_cache`
  `https://en.wikipedia.org/wiki/Cache_(computing)`

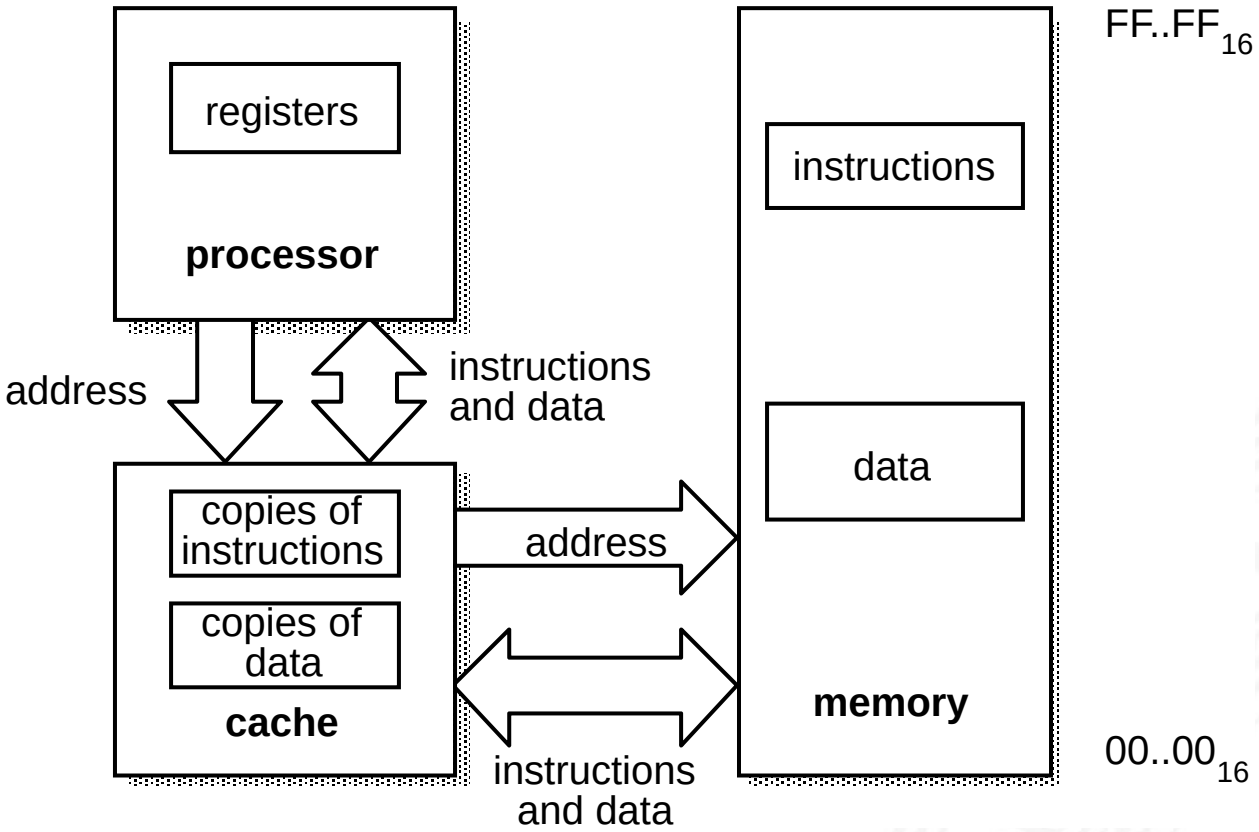
Cache (cont.)

- CPU referenziert Adresse
  - parallele Suche in L1 (level 1), L2 ... und Hauptspeicher
  - erfolgreiche Suche liefert Datum, Abbruch laufender Suchen



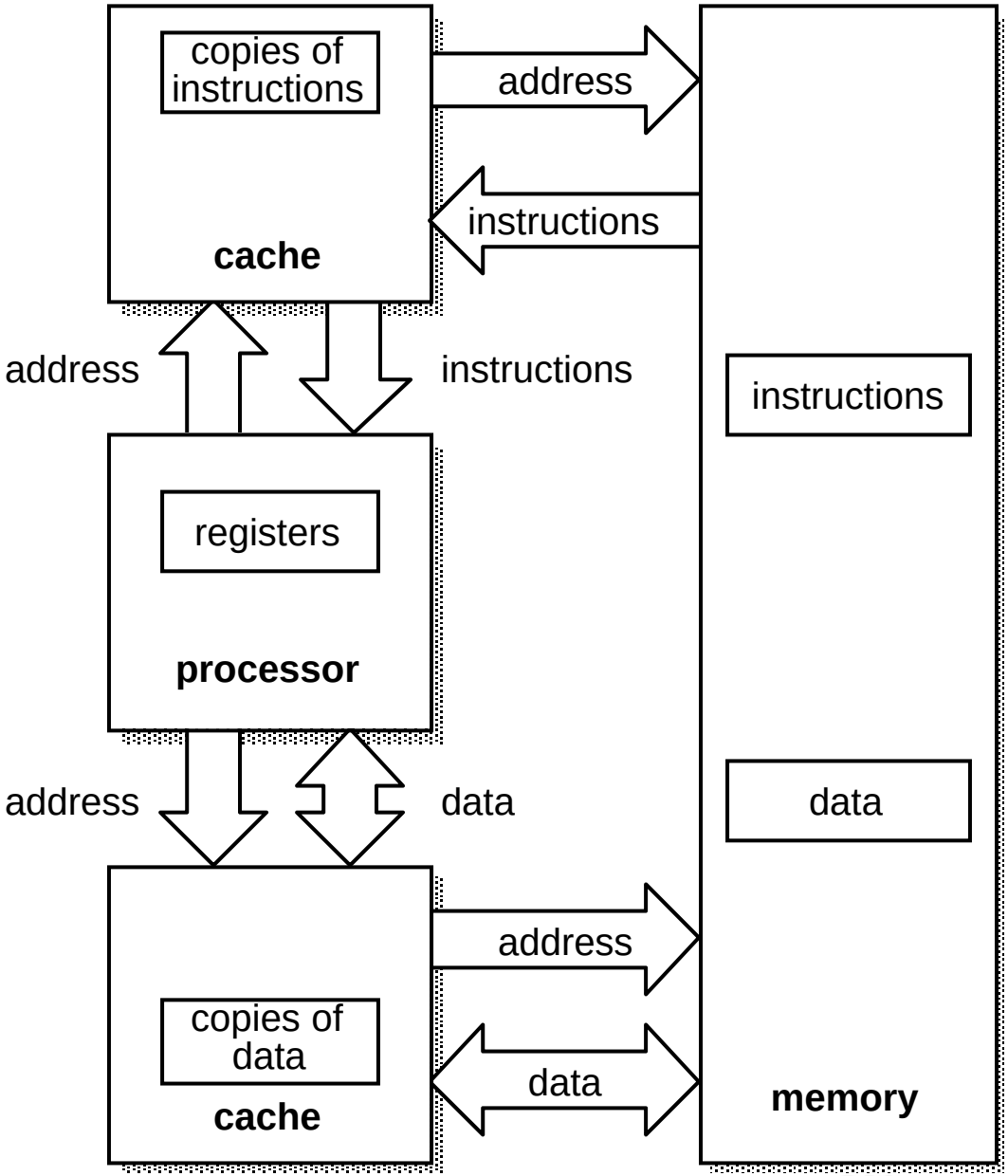
[BO15]

gemeinsamer Cache / „unified Cache“



[Fur00]

separate Instruction-/Data Caches



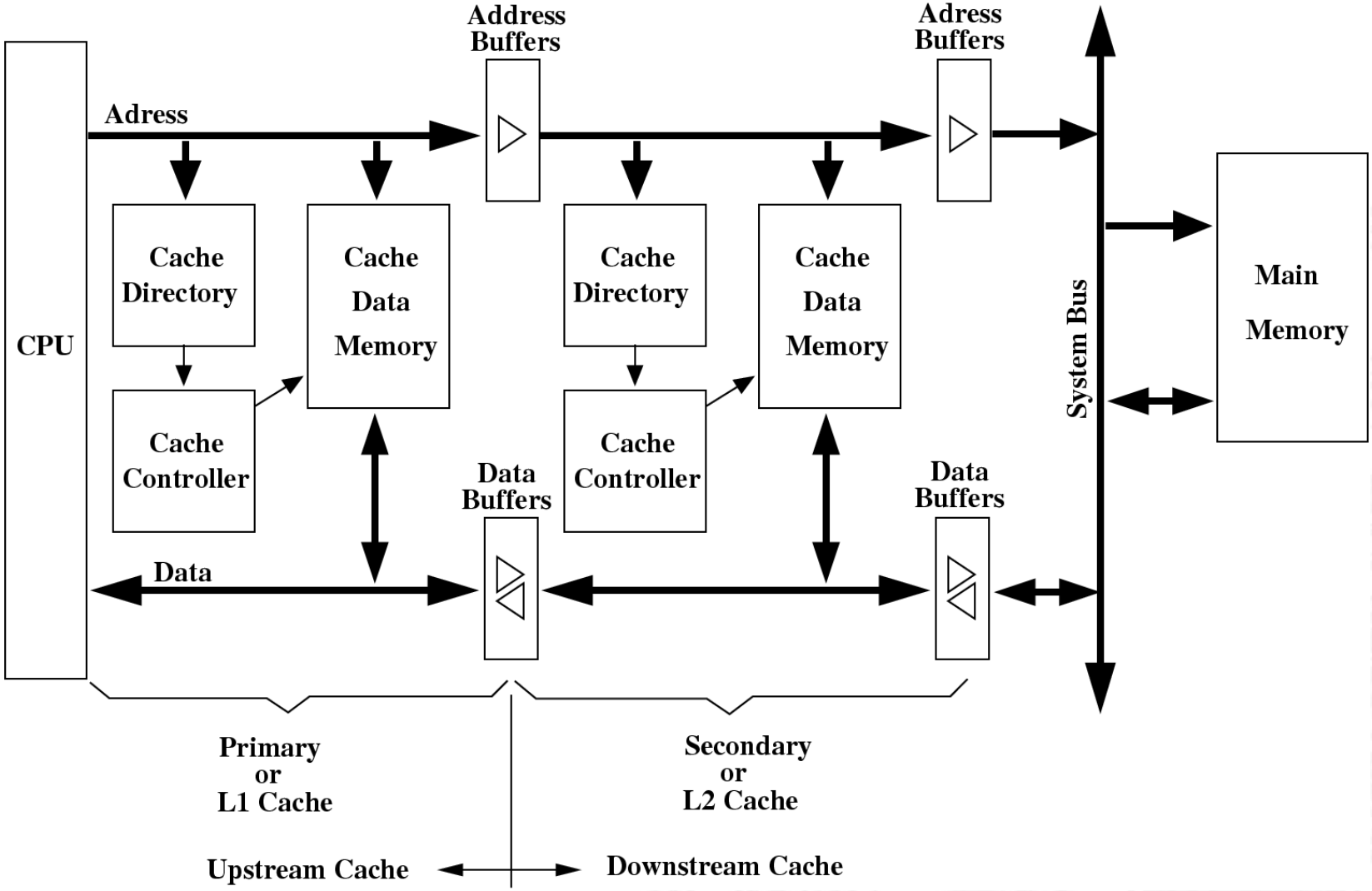
$FF..FF_{16}$

$00..00_{16}$

[Fur00]

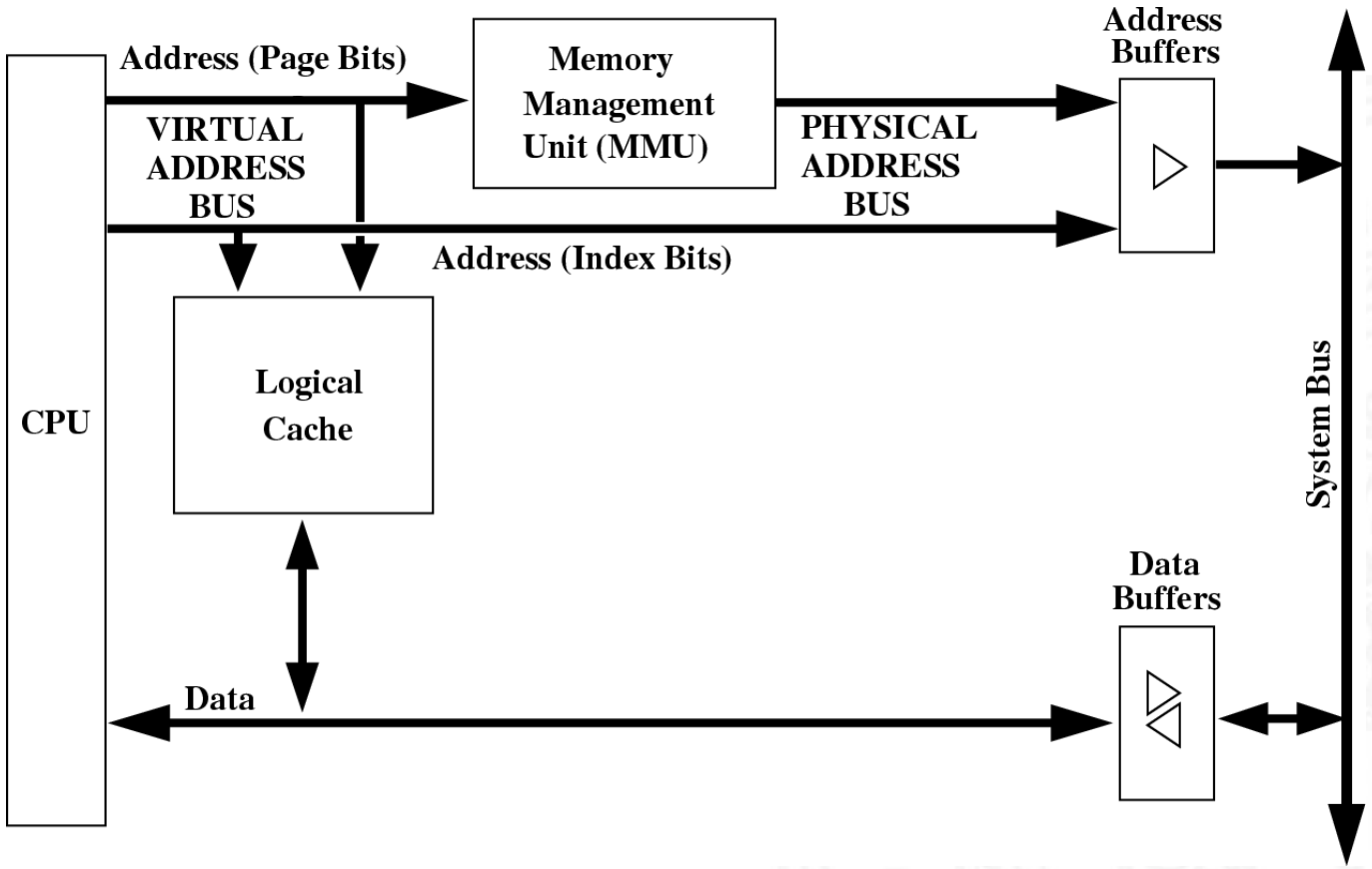
Cache – Position

- First- und Second-Level Cache



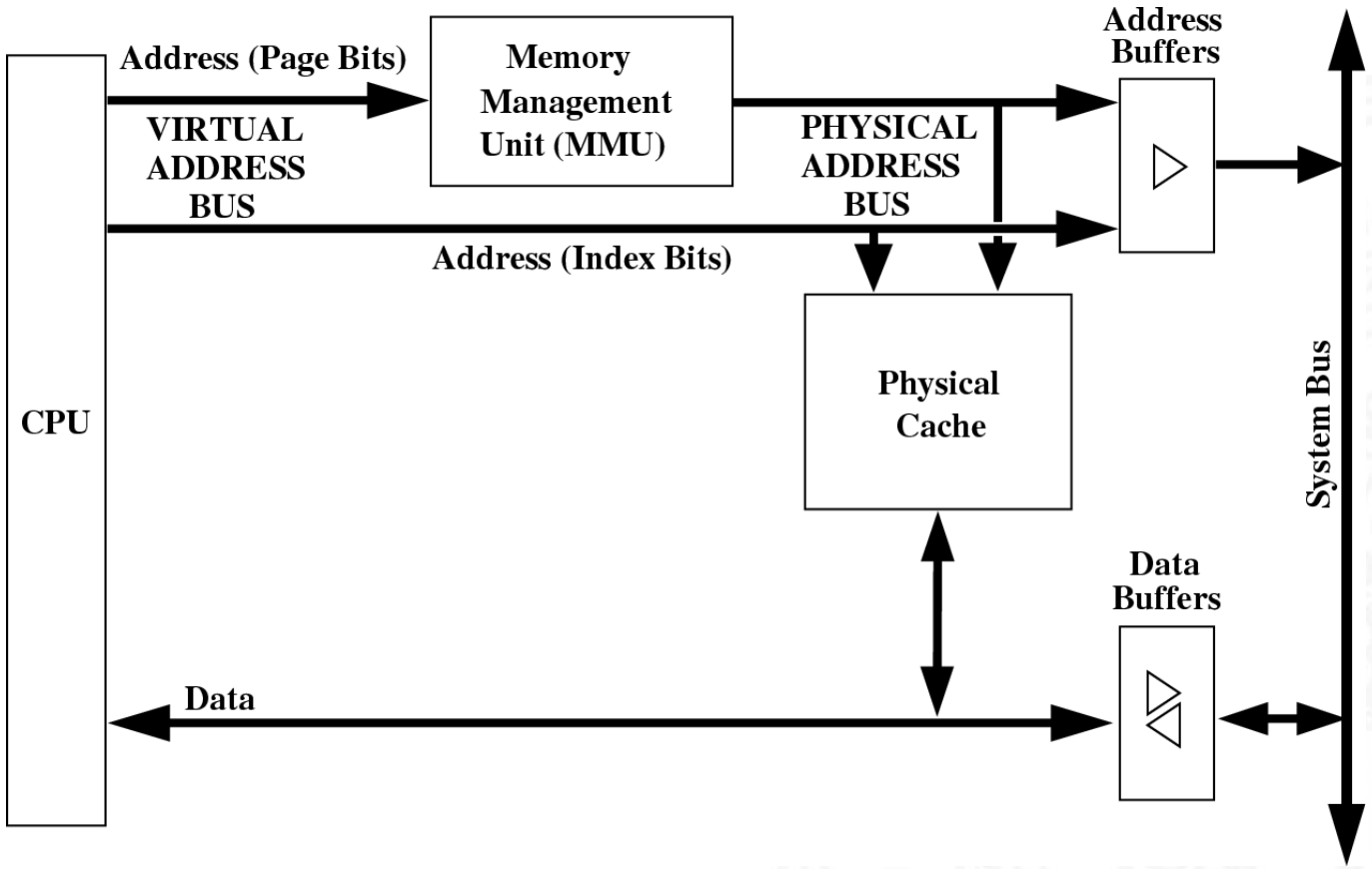
Cache – Position (cont.)

- Virtueller Cache
  - \+ Adressumrechnung durch MMU oft nicht nötig
  - – Cache leeren bei Kontextwechseln



Cache – Position (cont.)

- Physikalischer Cache
  - \+ Cache muss nie geleert werden
  - – Adressumrechnung durch MMU immer nötig



Cache – Position (cont.)

- typische Cache Organisation
  - First-Level Cache: getrennte Instruktions- und Daten-Caches
  - Second-Level Cache: gemeinsamer Cache je Prozessorkern
  - Third-Level Cache: gemeinsamer Cache für alle Prozessorkerne
- bei mehreren Prozessoren / Prozessorkernen

  $\Rightarrow$ Cache-Kohärenz wichtig
  - gemeinsam genutzte Daten konsistent halten (s.u.)

Cache – Strategie

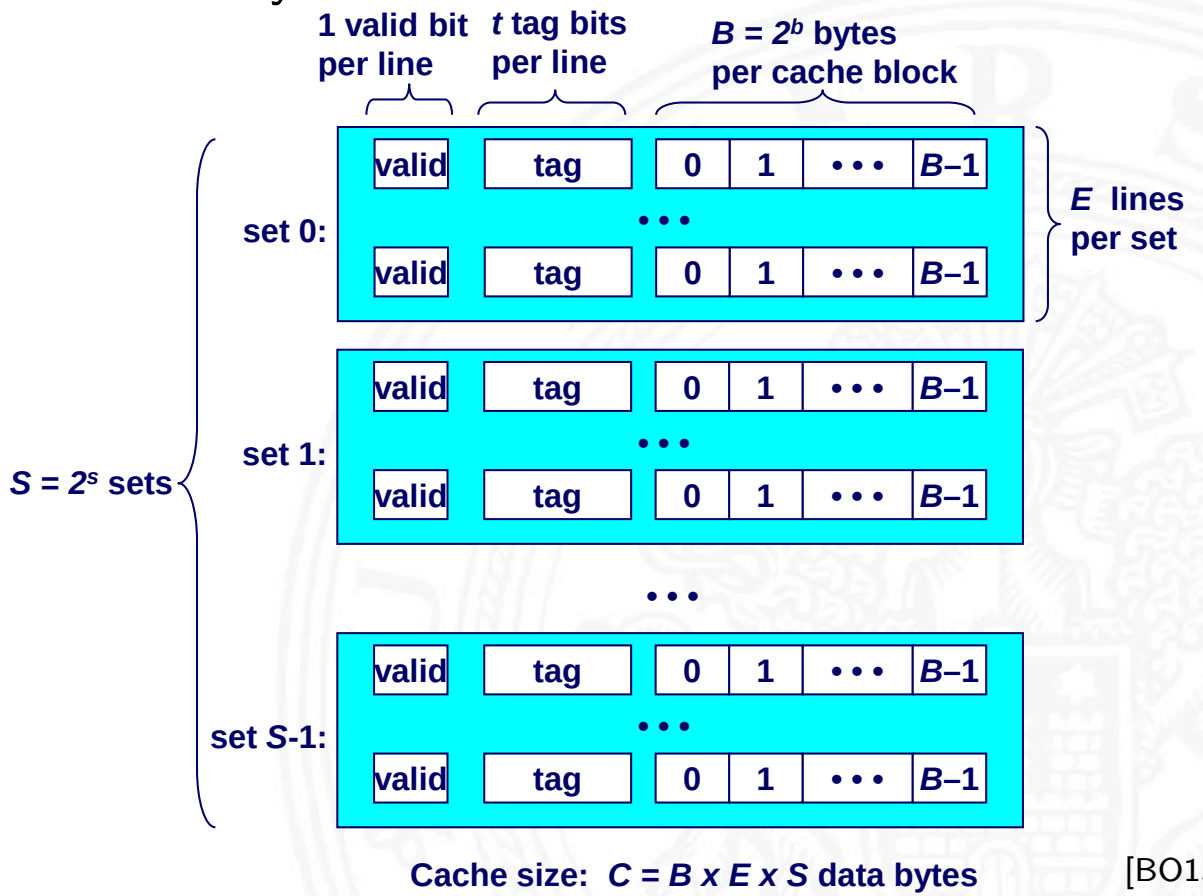
Cachestrategie: *Welche Daten sollen in den Cache?*
Diejenigen, die bald wieder benötigt werden!

- *temporale Lokalität*:
  die Daten, die zuletzt häufig gebraucht wurden
- *räumliche Lokalität*:
  die Daten, die nahe den zuletzt gebrauchten liegen

- verschiedene Platzierungs-, Ersetzungs- und Rückschreibestategien für den Cache

Cache Organisation

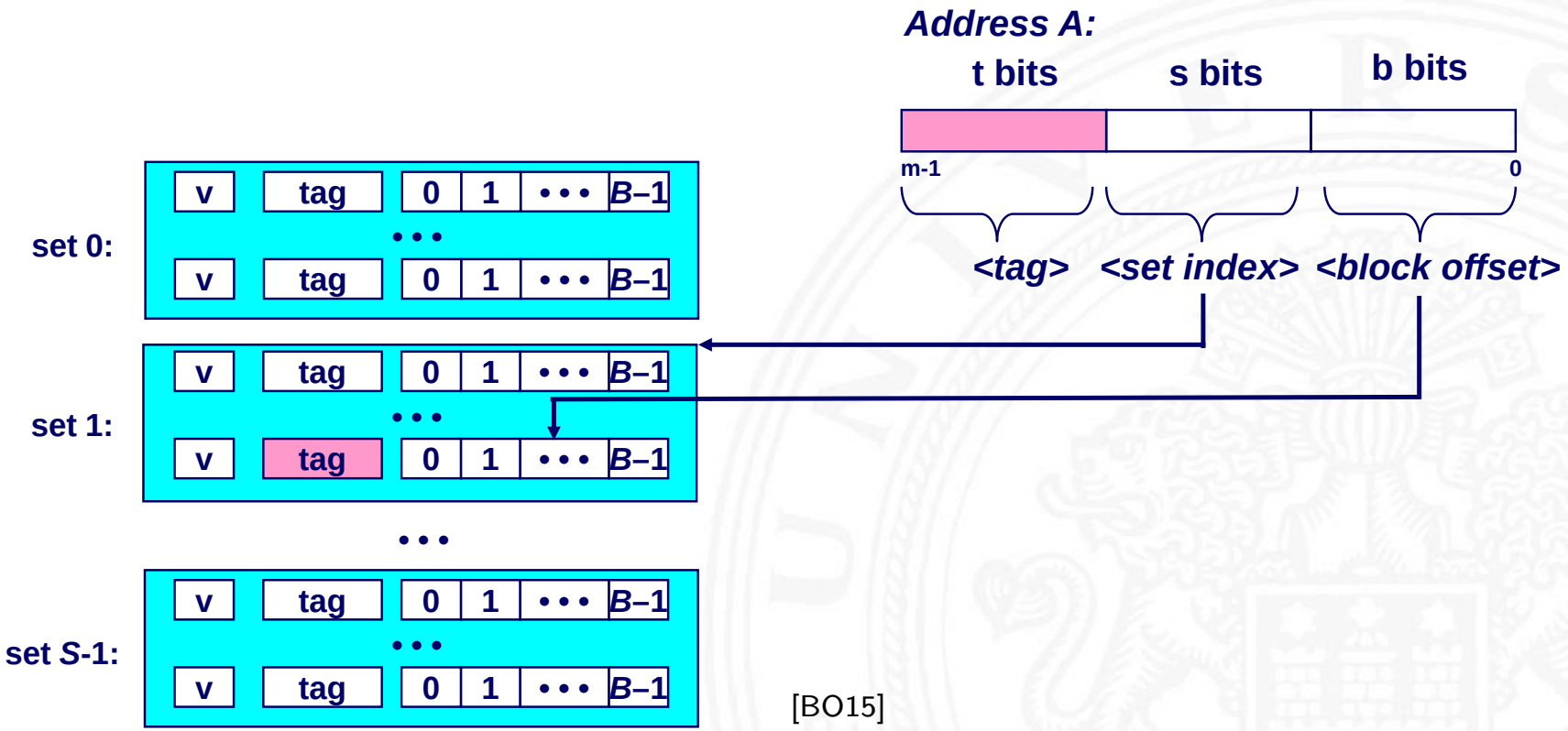
- Cache ist ein Array von Speicher-Bereichen („sets“)
- jeder Bereich enthält eine oder mehrere Zeilen
- jede Zeile enthält einen Datenblock
- jeder Block enthält mehrere Byte



[BO15]

Adressierung von Caches

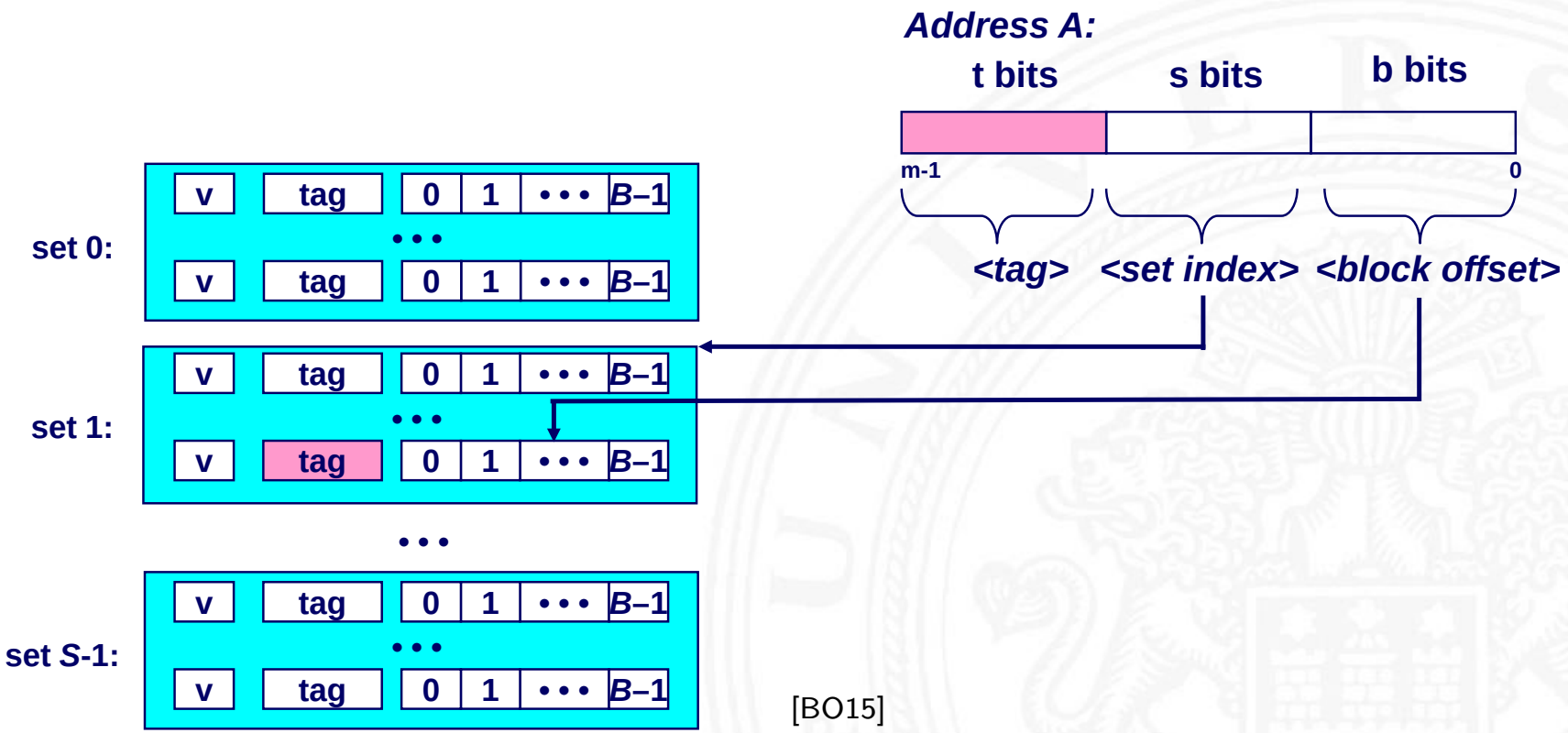
- Adressteil $\langle set\ index \rangle$ von A bestimmt Bereich („set“)
- Adresse A ist im Cache, wenn
  1. Cache-Zeile ist als gültig markiert („valid“)
  2. Adressteil $\langle tag \rangle$ von A = „tag“ Bits des Bereichs



[BO15]

Adressierung von Caches (cont.)

- Cache-Zeile („cache line“) enthält Datenbereich von 2^b Byte
- gesuchtes Wort mit Offset $\langle \textit{block offset} \rangle$



[BO15]

Cache – Organisation

- *Welchen Platz im Cache belegt ein Datum des Hauptspeichers?*
- drei Verfahren

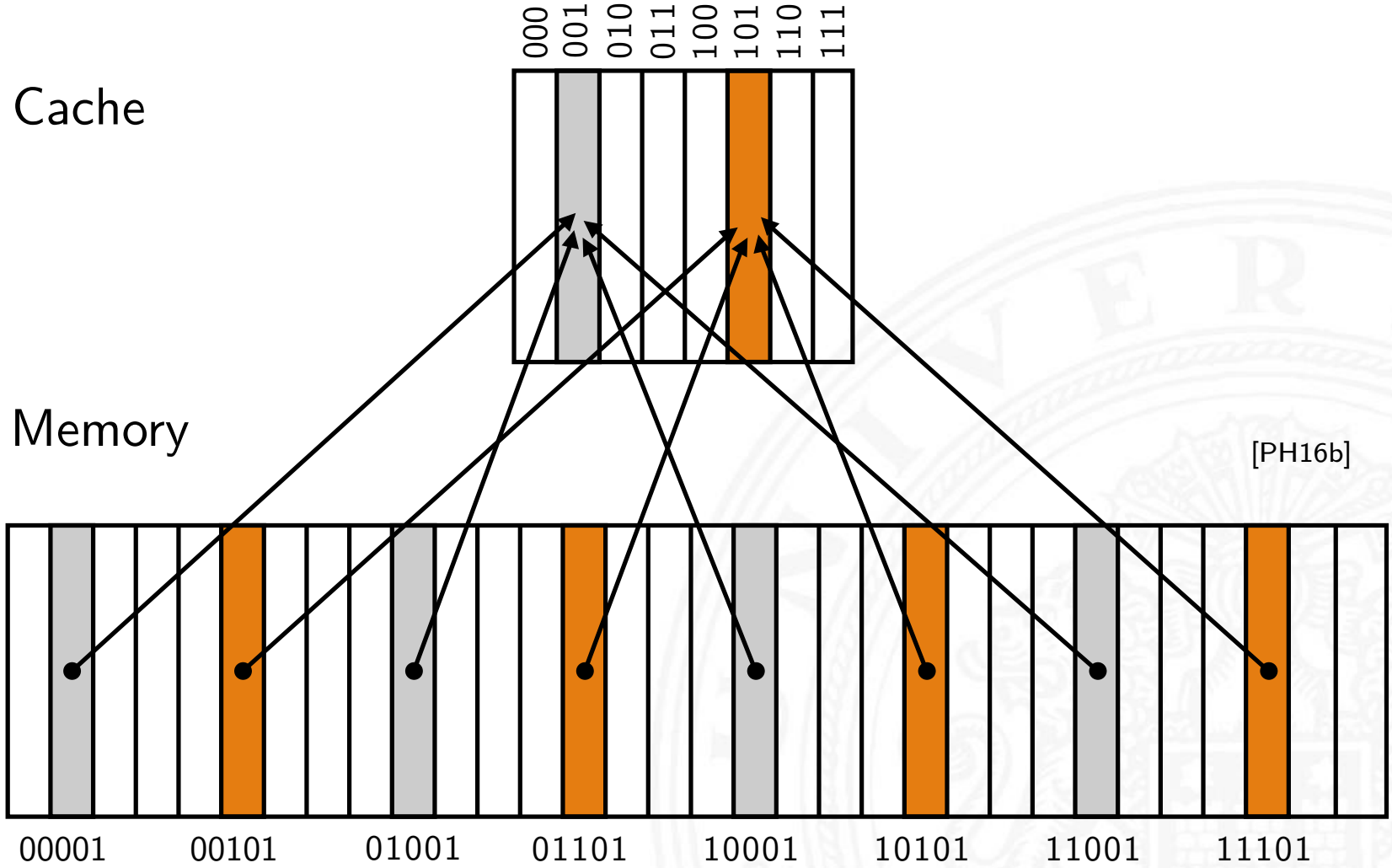
direkt abgebildet / direct mapped jeder Speicheradresse ist genau eine Cache-Speicherzelle zugeordnet

n-fach bereichsassoziativ / set associative
jeder Speicheradresse ist eine von E möglichen Cache-Speicherzellen zugeordnet

voll-assoziativ jeder Speicheradresse kann jede beliebige Cache-Speicherzelle zugeordnet werden

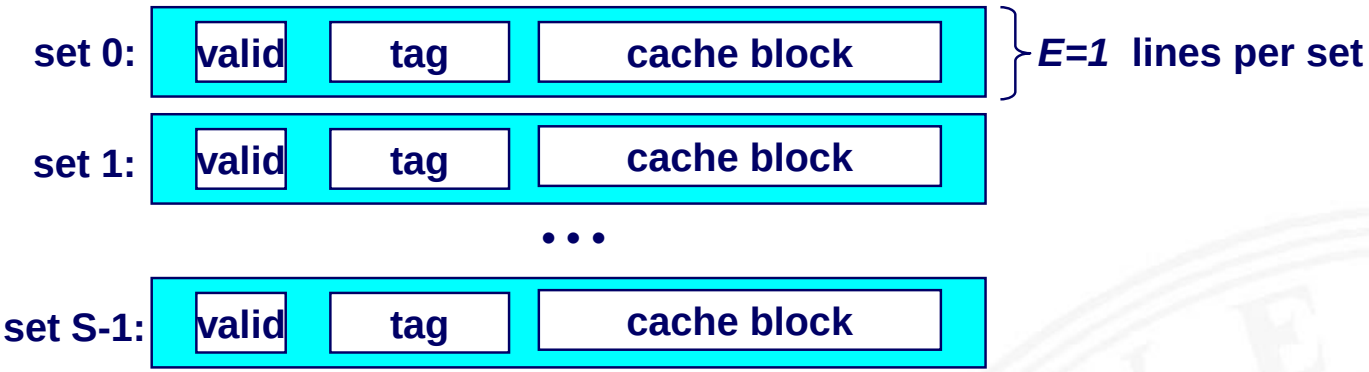
Cache: direkt abgebildet / „direct mapped“

- jeder Adresse ist genau eine Speicherzelle im Cache zugeordnet



Cache: direkt abgebildet / „direct mapped“ (cont.)

- verfügt über genau 1 Zeile pro Bereich S Bereiche (**S**ets)



[BO15]

\+ einfachste Cache-Art

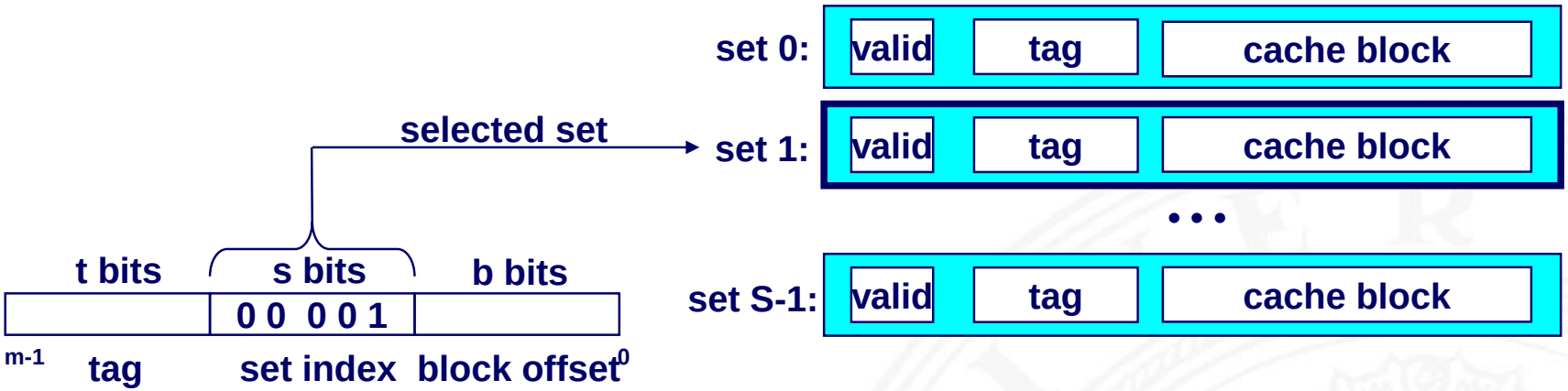
\+ große Caches möglich

– Effizienz, z.B. Zugriffe auf A, $A + n \cdot S \ldots$
$\Rightarrow$ „*Cache Thrashing*“

Cache: direkt abgebildet / „direct mapped“ (cont.)

Zugriff auf direkt abgebildete Caches

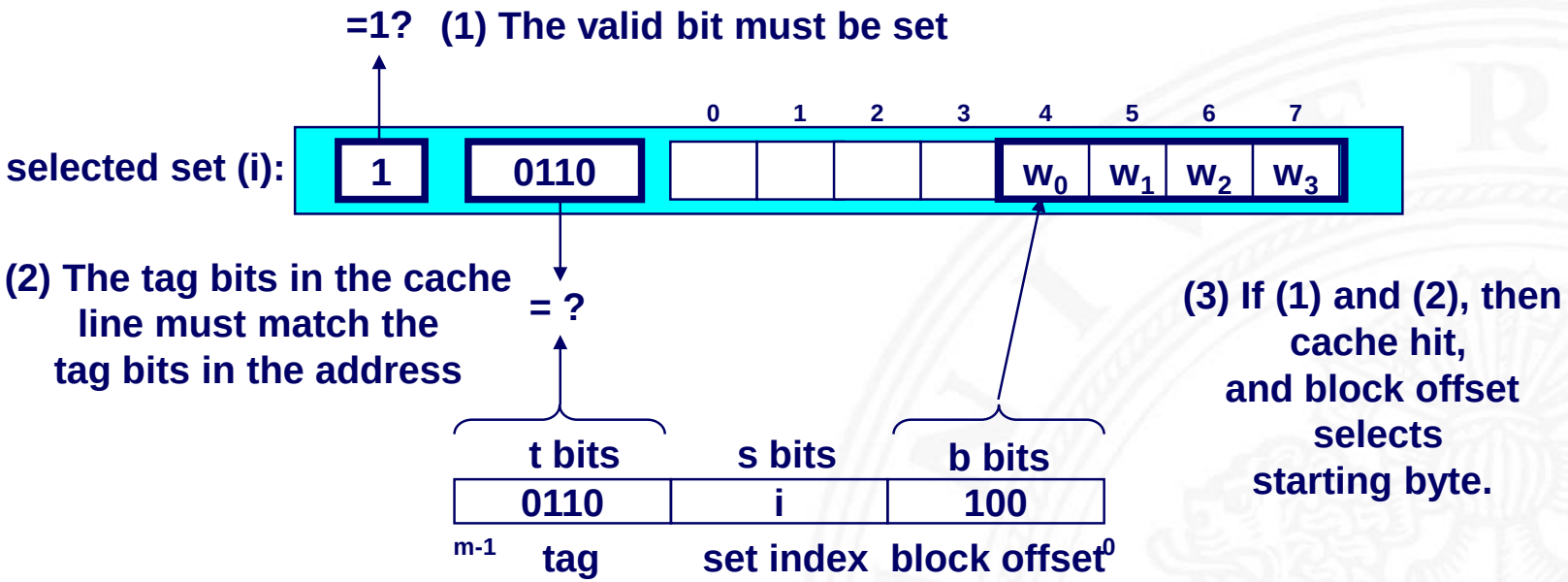
1. Bereichsauswahl durch Bits ⟨*set index*⟩



[BO15]

Cache: direkt abgebildet / „direct mapped“ (cont.)

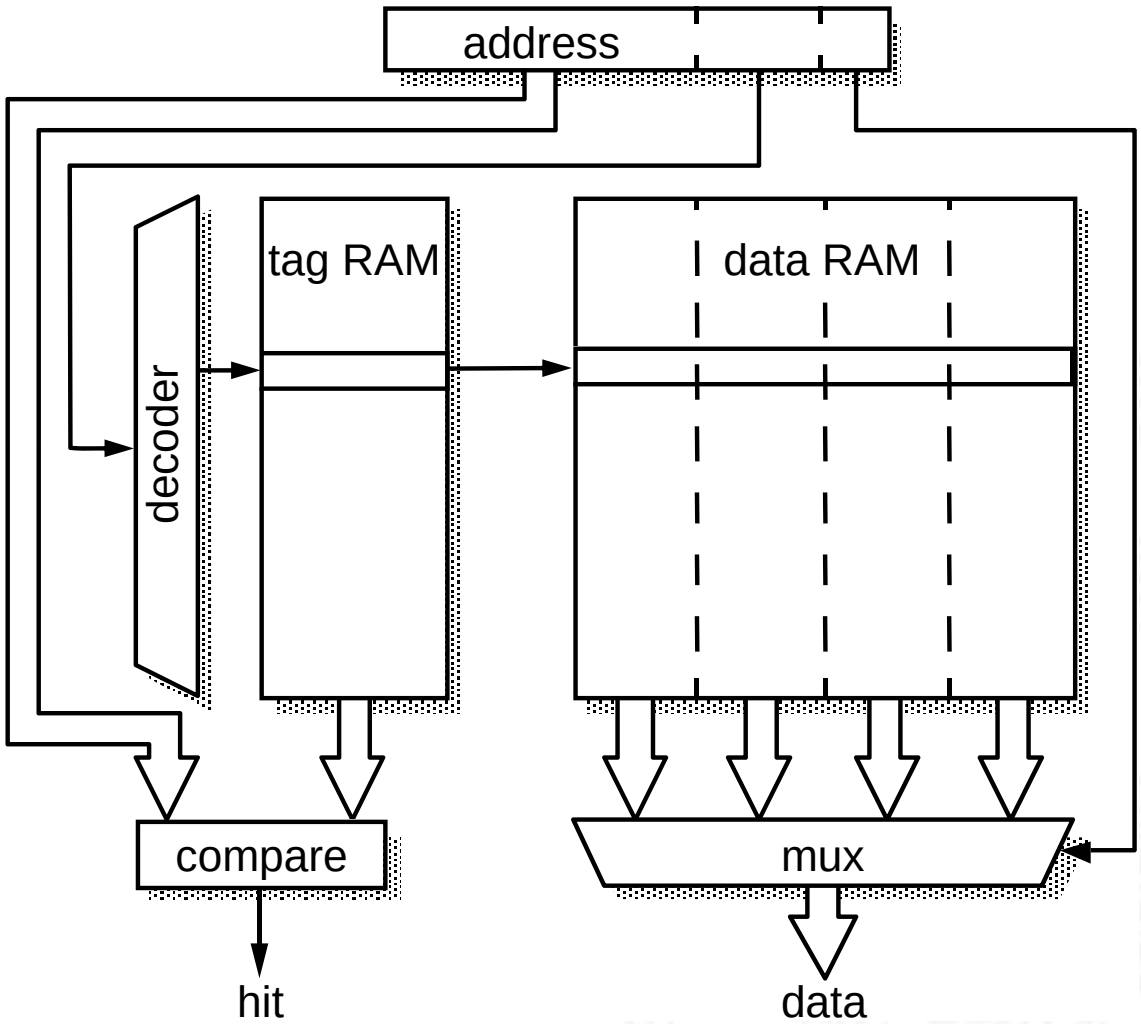
2. ⟨*valid*⟩: sind die Daten gültig?
3. „Line matching“: stimmt ⟨*tag*⟩ überein?
4. Wortselektion extrahiert Wort unter Offset ⟨*block offset*⟩



[BO15]

Cache: direkt abgebildet / „direct mapped“ (cont.)

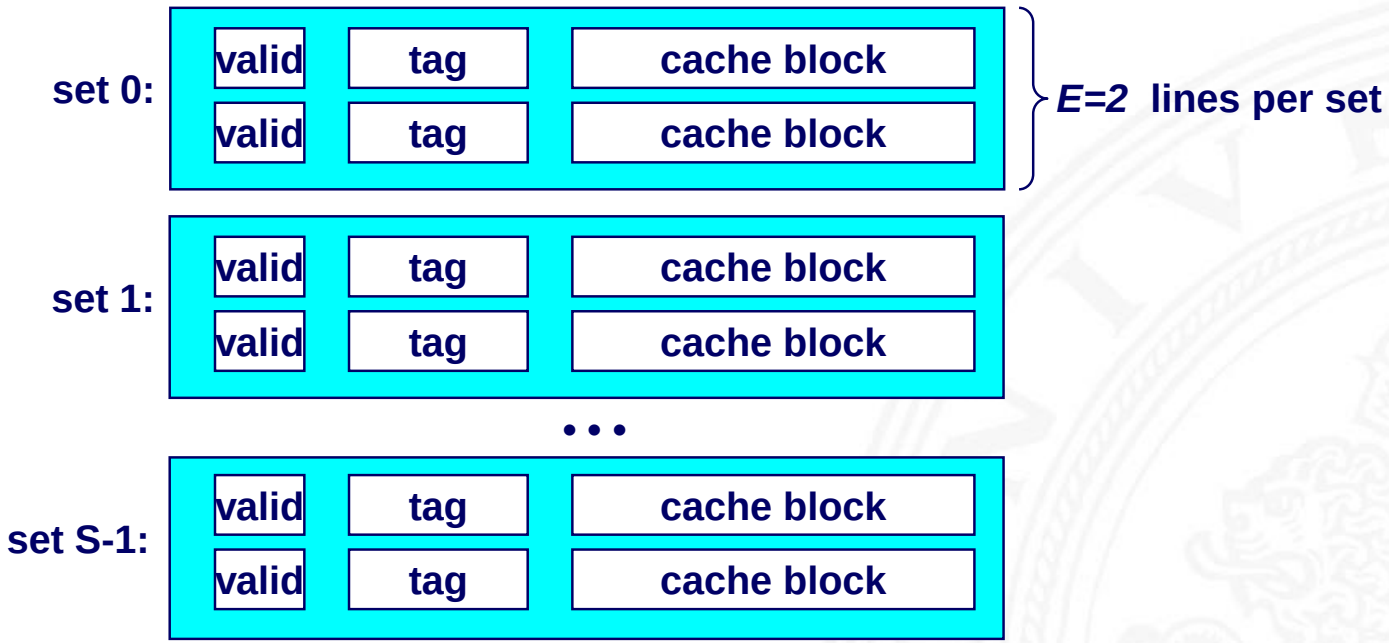
Prinzip



[Fur00]

Cache: bereichsassoziativ / „set associative“

- jeder Speicheradresse ist ein Bereich S mit mehreren (E) Cachezeilen zugeordnet
- n-fach assoziative Caches: E=2, 4 ...
  „2-way set associative cache“, „4-way ...“

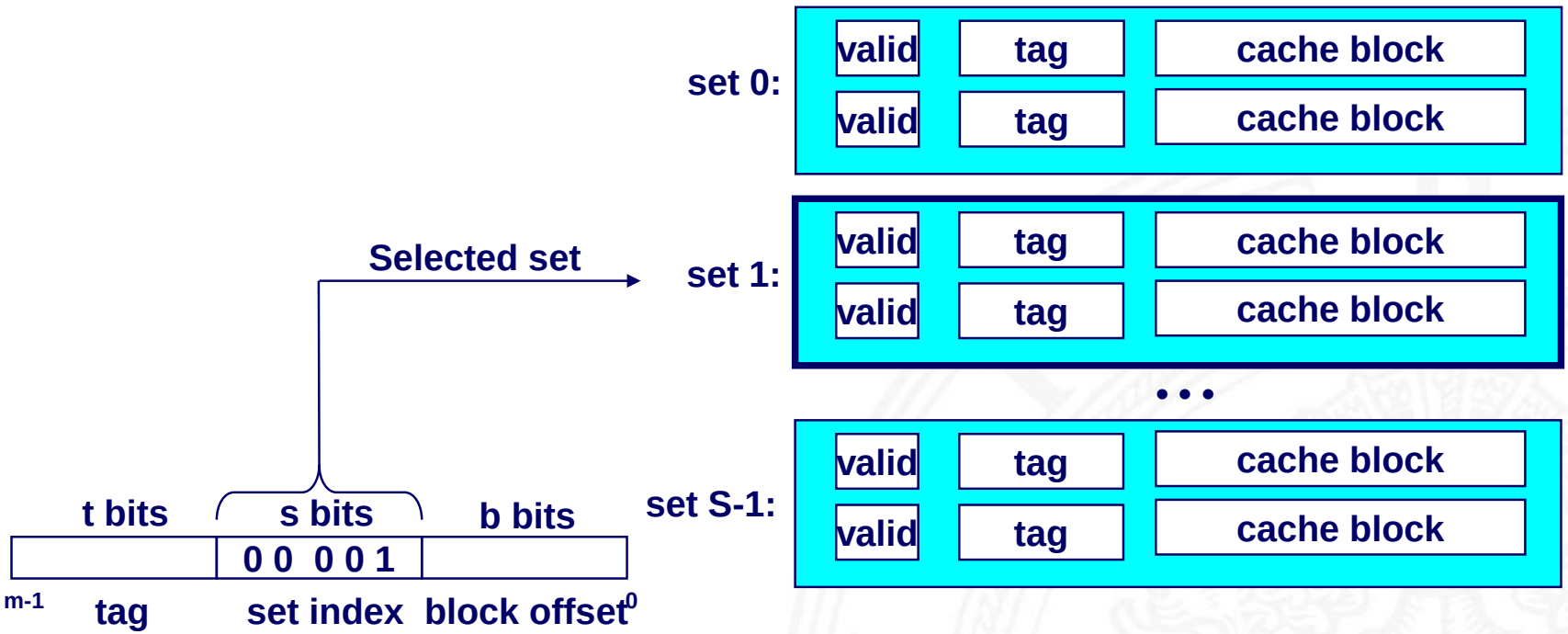


[BO15]

Cache: bereichsassoziativ / „set associative“ (cont.)

Zugriff auf n-fach assoziative Caches

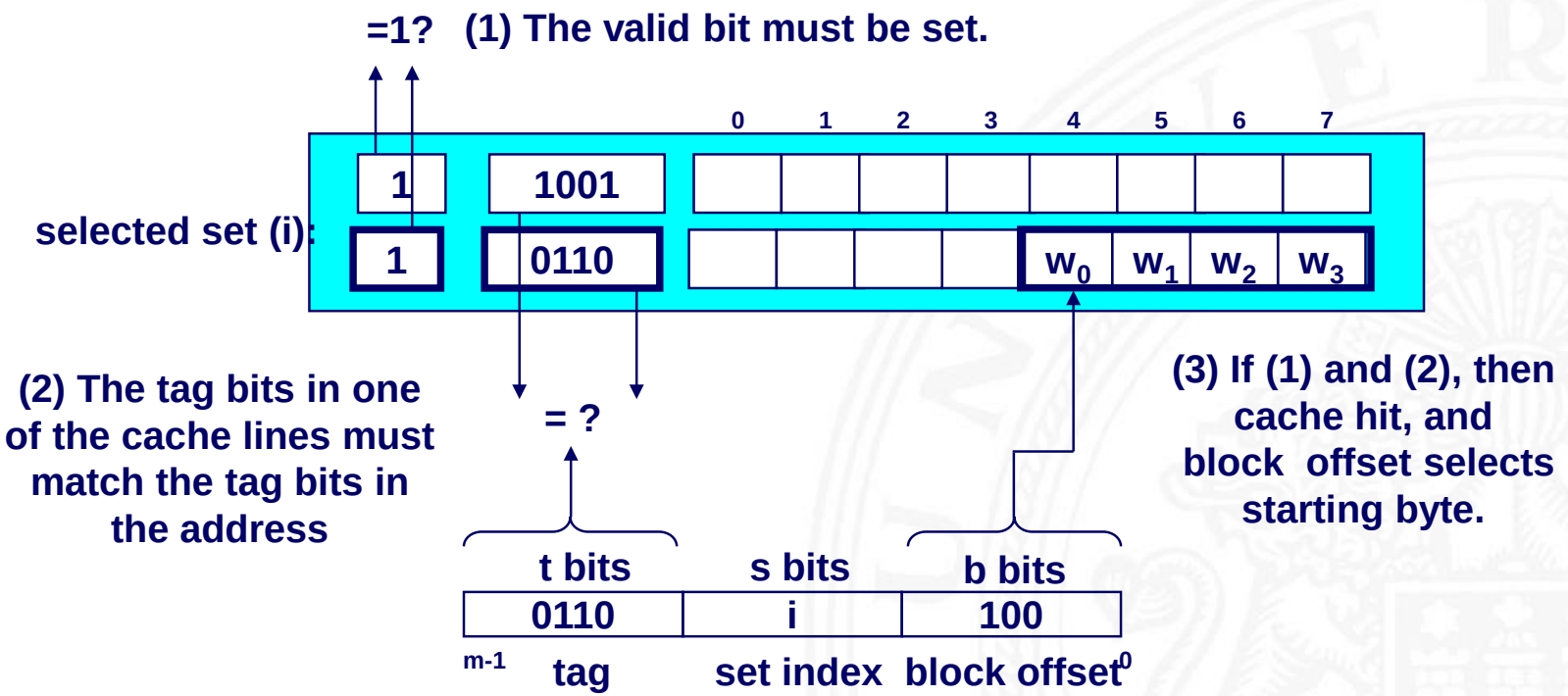
1. Bereichsauswahl durch Bits ⟨*set index*⟩



[BO15]

Cache: bereichsassoziativ / „set associative“ (cont.)

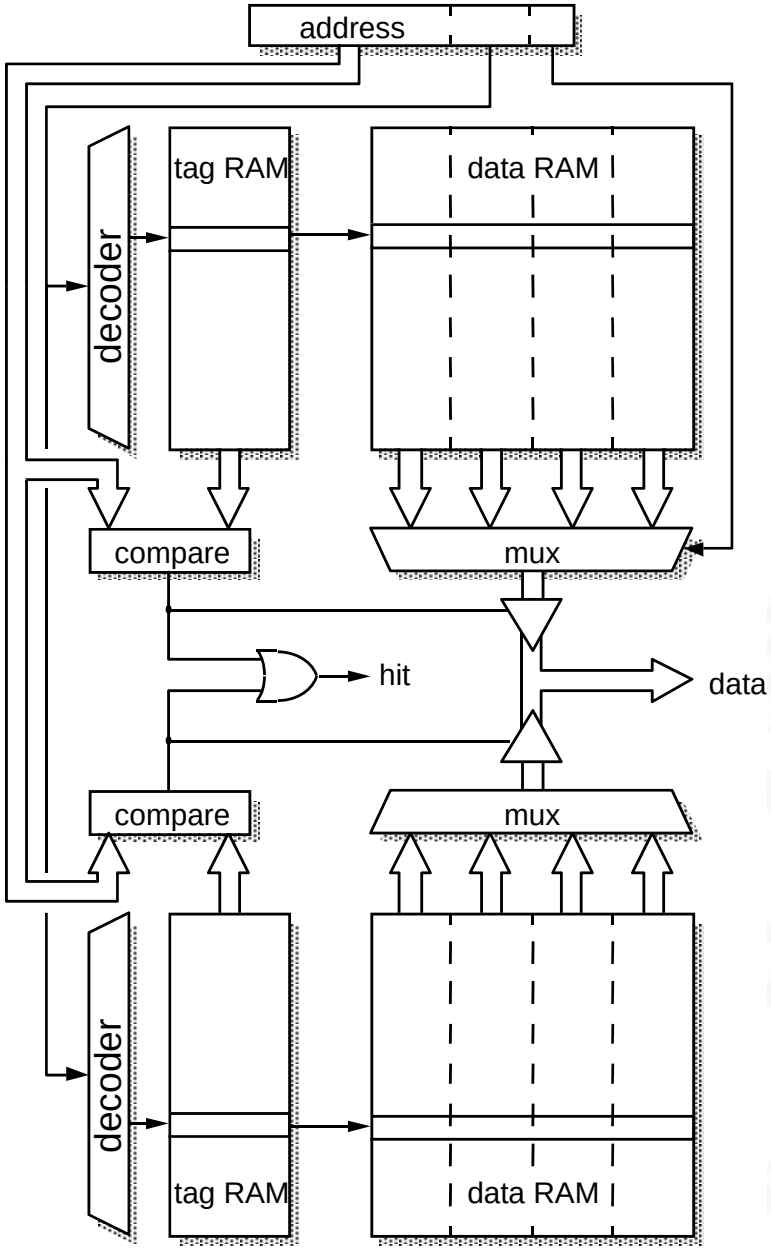
2. ⟨*valid*⟩: sind die Daten gültig?
3. „Line matching“: Cache-Zeile mit passendem ⟨*tag*⟩ finden?
   dazu Vergleich aller „tags“ des Bereichs ⟨*set index*⟩
4. Wortselektion extrahiert Wort unter Offset ⟨*block offset*⟩



[BO15]

Cache: bereichsassoziativ / „set associative“ (cont.)

Prinzip



[Fur00]

Cache: voll-assoziativ

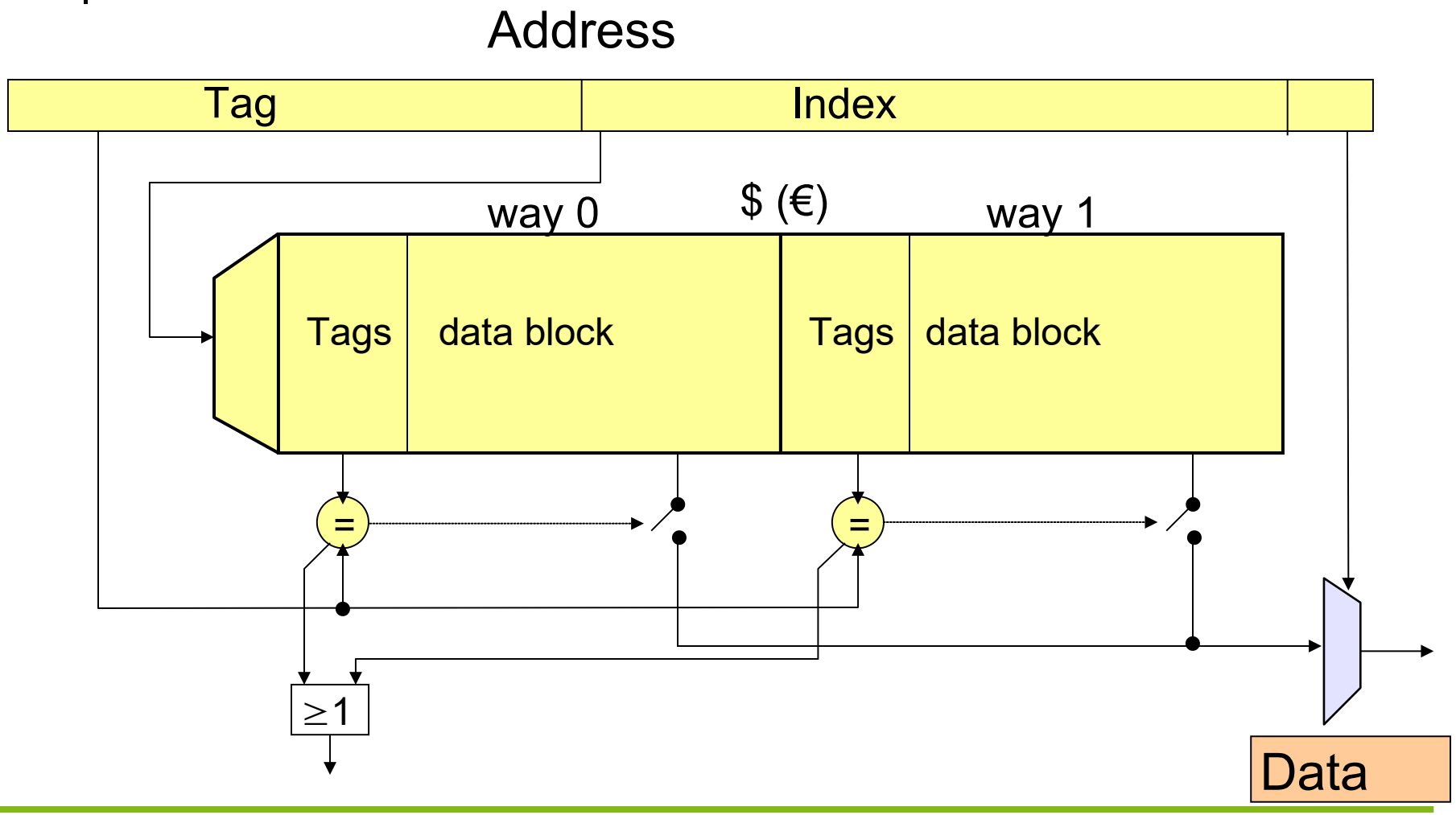
- jeder Adresse des Speichers kann jede beliebige Cachezeile zugeordnet werden
- Spezialfall: nur ein Cachebereich S

– benötigt E-Vergleicher
– nur für sehr kleine Caches realisierbar

– Ende –

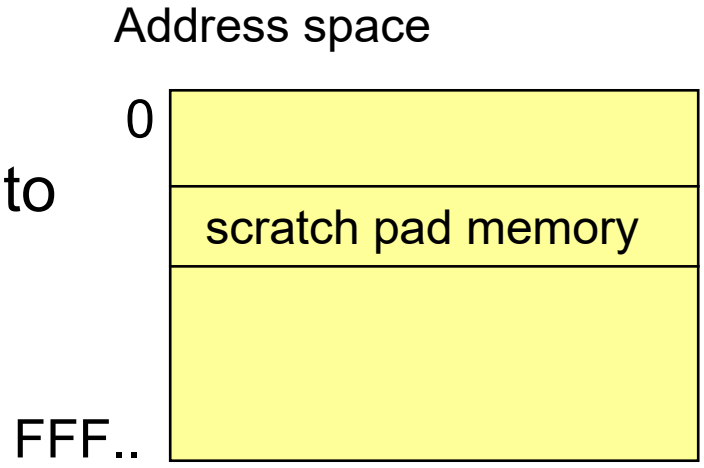
Set-associative cache n-way cache

|Set| = 2



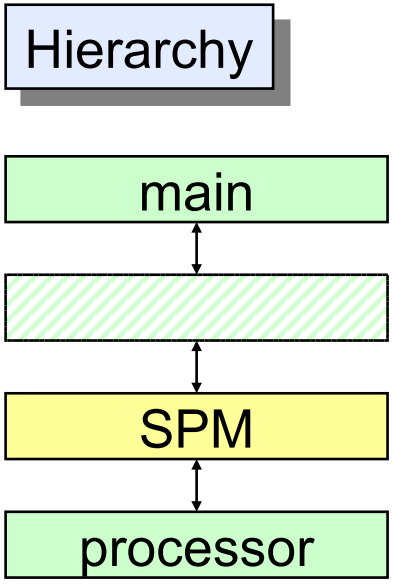
Hierarchical memories using scratch pad memories (SPM)

SPM is a small, physically separate memory mapped into the address space

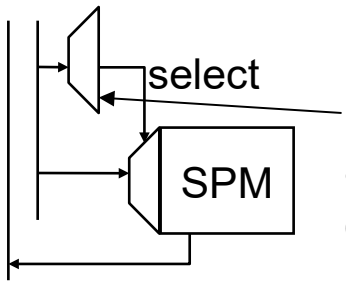


Examples:

- Most ARM cores allow tightly coupled memories
- IBM Cell
- Infineon TriCore
- Many multi-cores, due to high costs of coherent caches



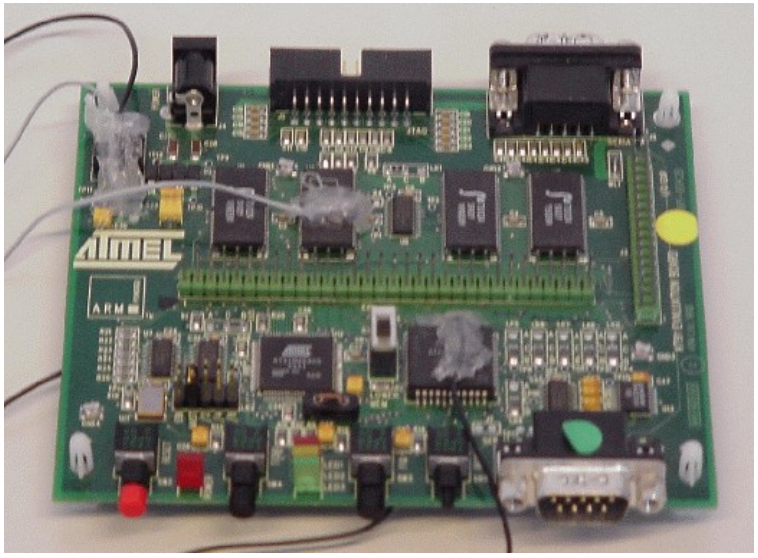
no tag memory

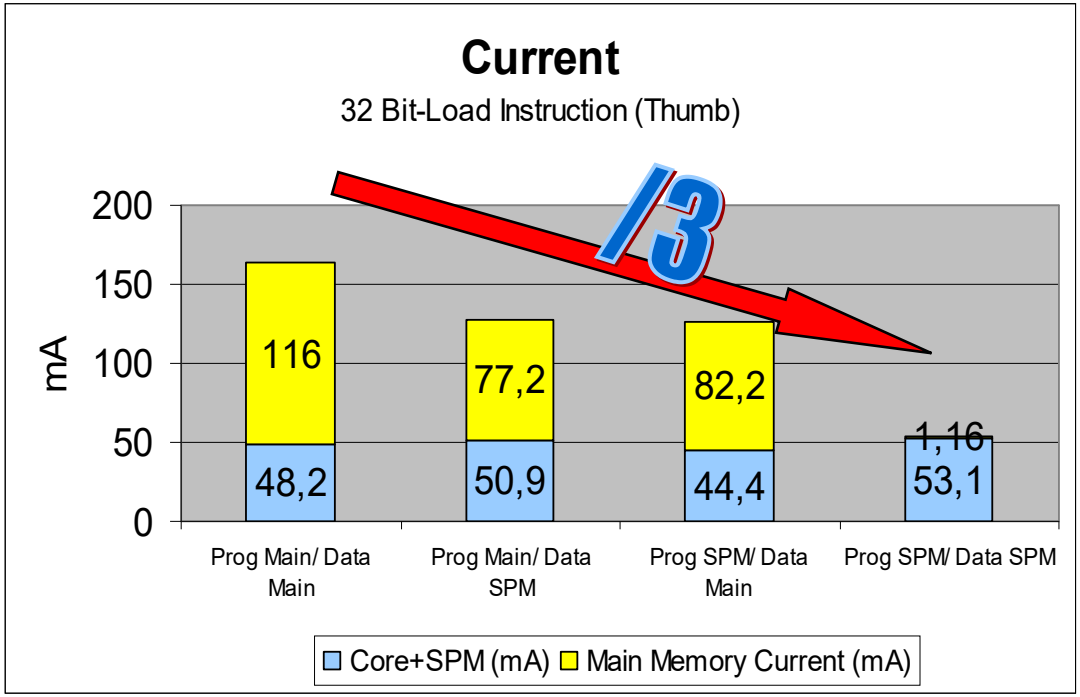


Selection is by an appropriate address decoder (simple!)

Comparison of currents using measurements

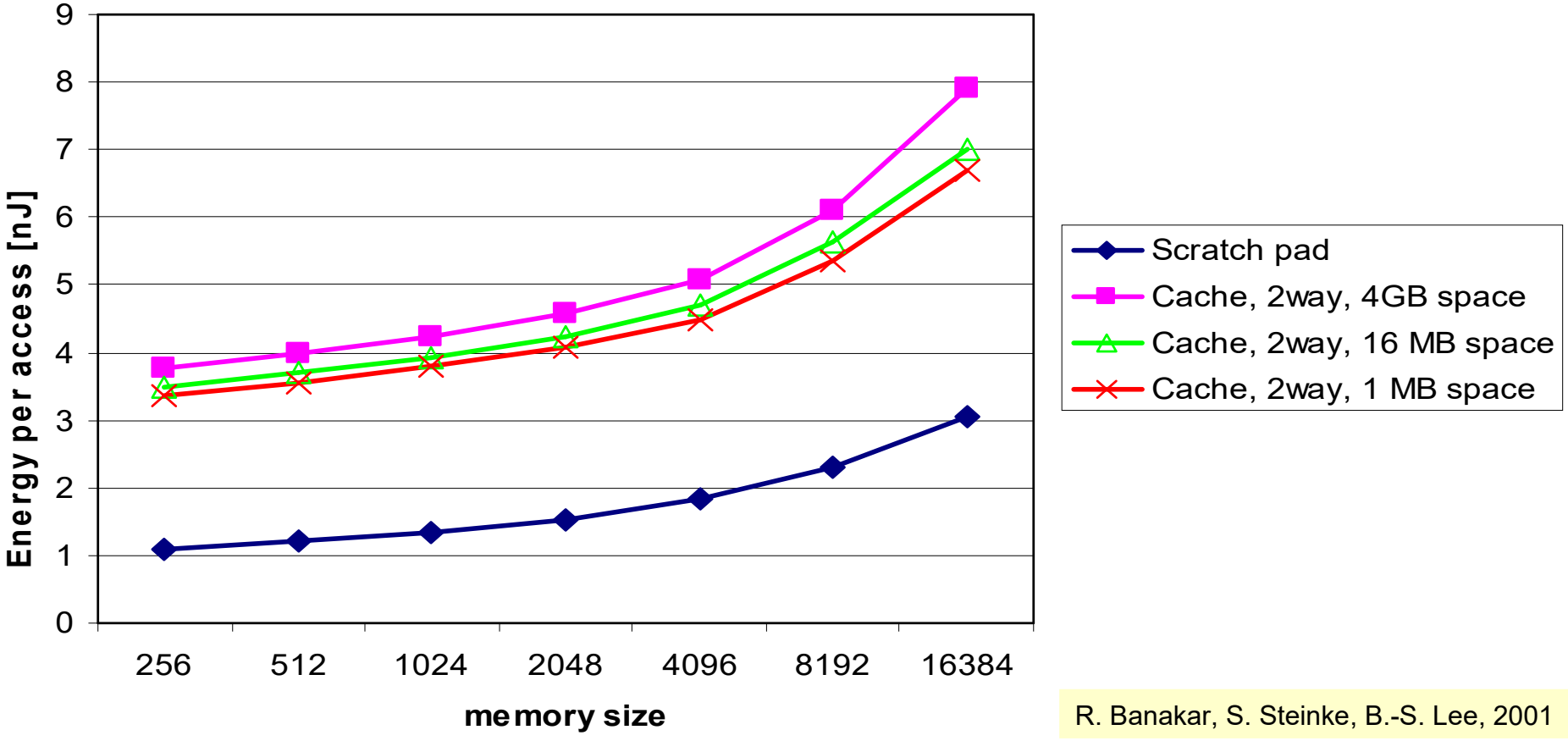
E.g.: ATMEL board with ARM7TDMI and ext. SRAM





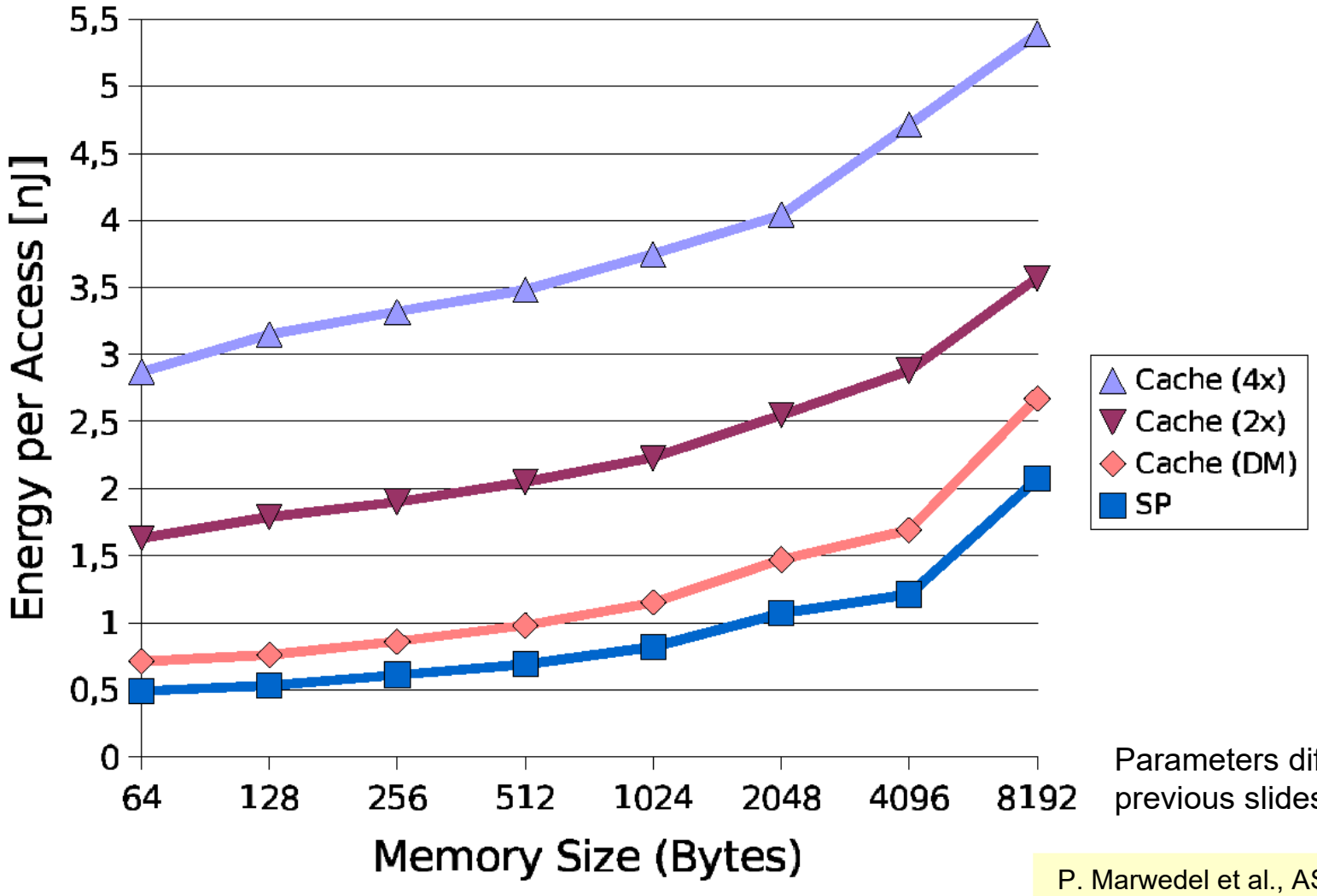
Why not just use a cache ?

Energy for parallel access of sets, in comparators, muxes.



R. Banakar, S. Steinke, B.-S. Lee, 2001

Influence of the associativity



Parameters different from previous slides

P. Marwedel et al., ASPDAC, 2004

Summary

- Processing
  - VLIW/EPIC processors
  - MPSoCs
- FPGAs
- Memories
  - “Small is beautiful”
    (in terms of energy consumption, access times, size)

Einschub: Pipelining

– Beginn –

von-Neumann Befehlszyklus

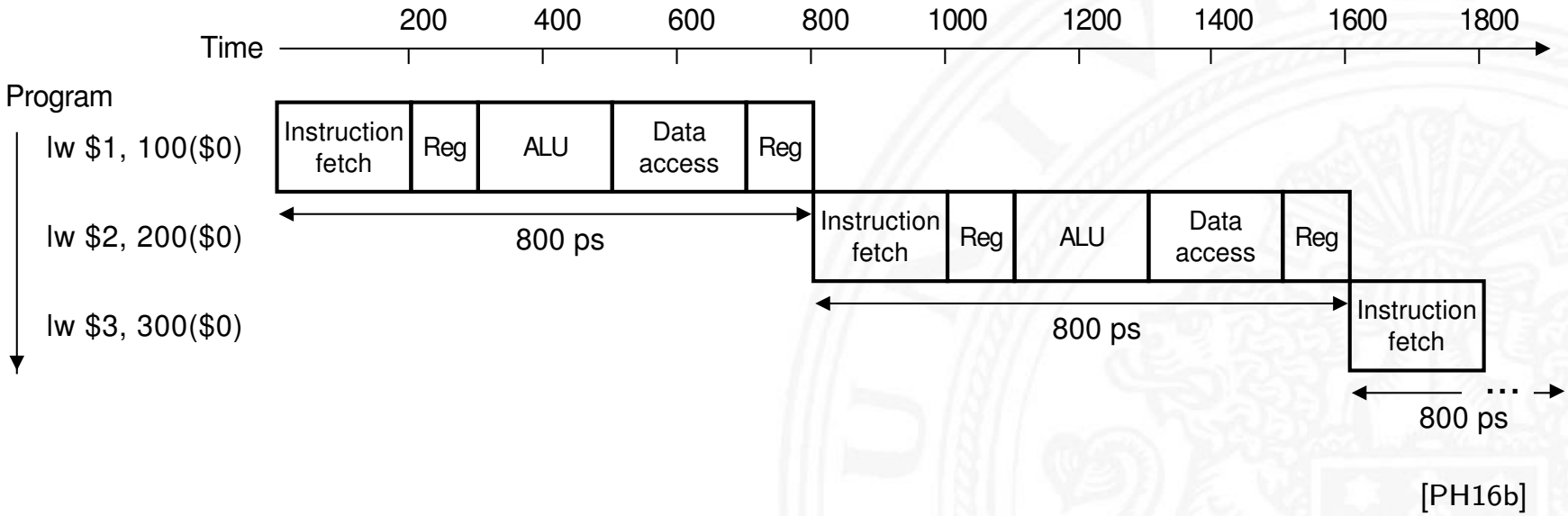
typische Schritte der Befehlsabarbeitung (von ISA abhängig)

- IF — **I**nstruction **F**etch
  Instruktion holen, in Befehlsregister laden
- ID — **I**nstruction **D**ecode
  Instruktion decodieren
- OF — **O**perand **F**etch
  Operanden aus Registern holen
- EX — **Ex**ecute
  ALU führt Befehl aus
- MEM — **Mem**ory access
  Speicherzugriff: Daten laden/abspeichern
- WB — **W**rite **B**ack
  Ergebnis in Register zurückschreiben

von-Neumann Befehlszyklus (cont.)

- je nach Instruktion sind 3-5 dieser Schritte notwendig
  - *nop*: nur Instruction-Fetch
  - *jump*: kein Speicher-/Registerzugriff
- Schritte können auch feiner unterteilt werden (mehr Stufen)

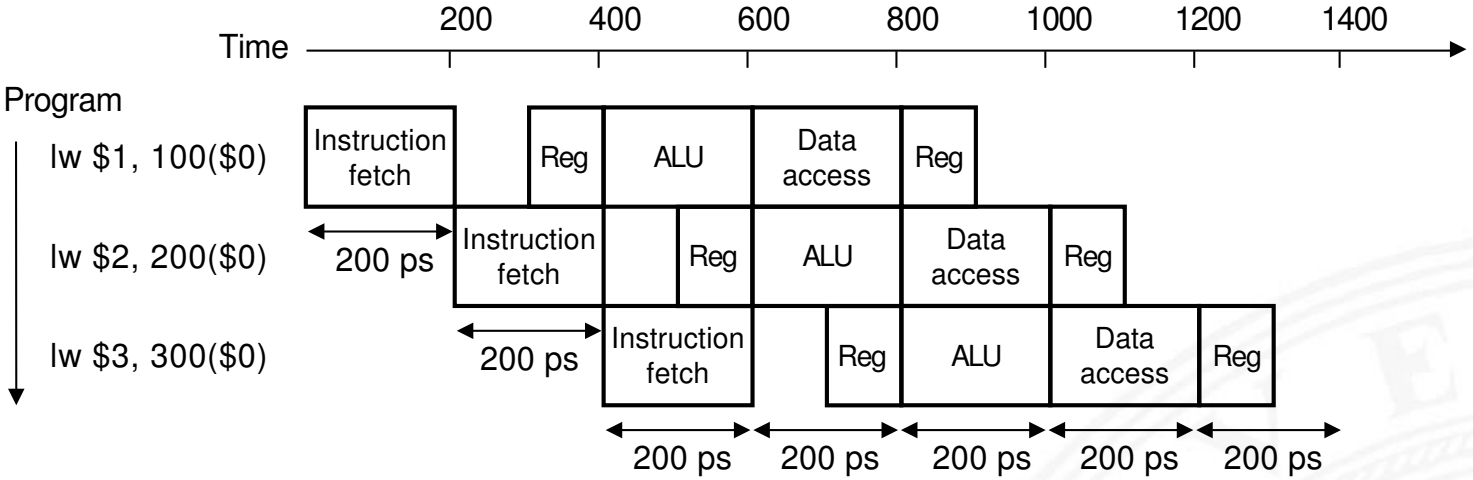
serielle Bearbeitung ohne Pipelining



[PH16b]

von-Neumann Befehlszyklus (cont.)

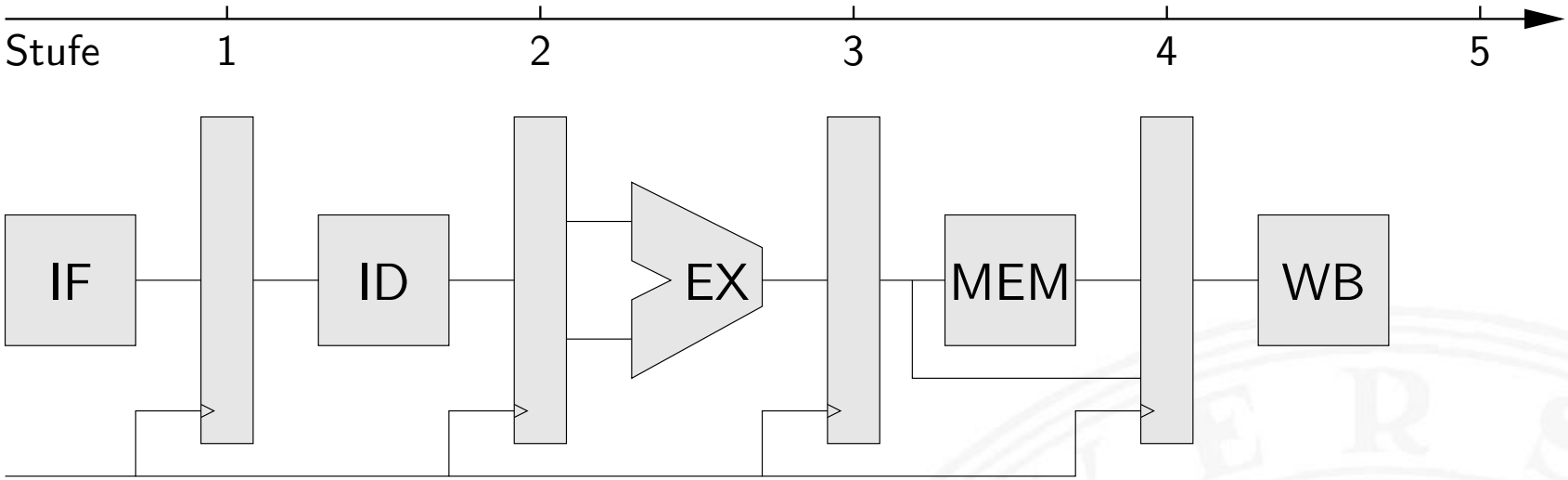
Pipelining für die einzelnen Schritte der Befehlsausführung



[PH16b]

- Befehle überlappend ausführen: neue Befehle holen, dann decodieren, während vorherige noch ausgeführt werden
- Register trennen Pipelinestufen

Klassische 5-stufige Pipeline



- Grundidee der ursprünglichen RISC-Architekturen
- \+ Durchsatz ca. 3 . . . 5 × besser als serielle Ausführung
- \+ guter Kompromiss aus Leistung und Hardwareaufwand

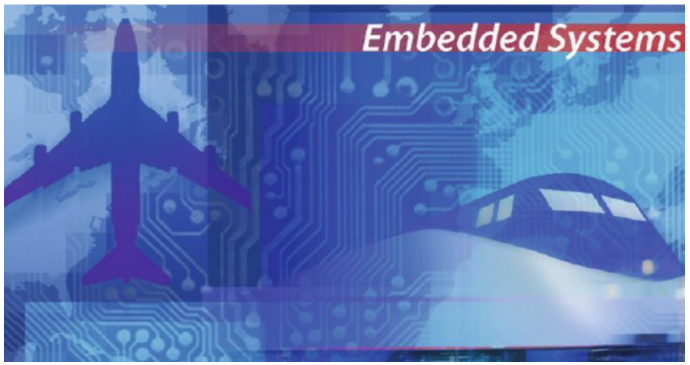
- MIPS-Architektur (aus Patterson, Hennessy [PH16b])

▸ MIPS ohne Pipeline ▸ MIPS Pipeline ▸ Pipeline Schema

– Ende –

Communication

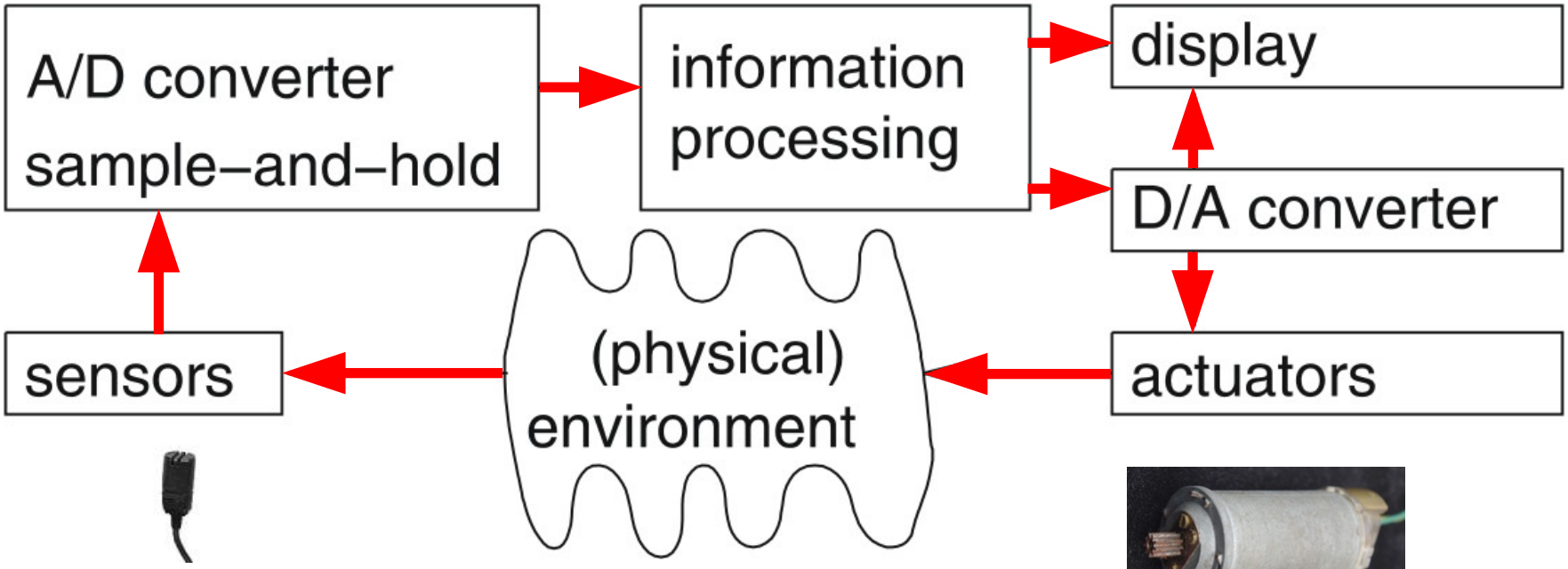
Peter Marwedel
TU Dortmund,
Informatik 12



2012 年 11 月 21 日

Embedded System Hardware

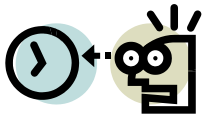
Embedded system hardware is frequently used in a loop (***"hardware in a loop"***):



☞ cyber-physical systems

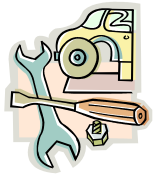
Communication
- Requirements -

- Real-time behavior



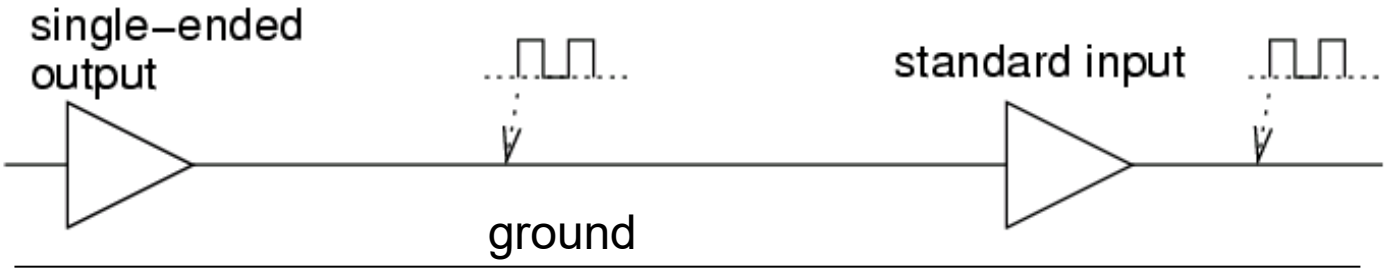
- Efficient, economical
  (e.g. centralized power supply)

- Appropriate bandwidth and communication delay
- Robustness
- Fault tolerance
- Diagnosability
- Maintainability
- Security
- Safety



Basic techniques: Electrical robustness

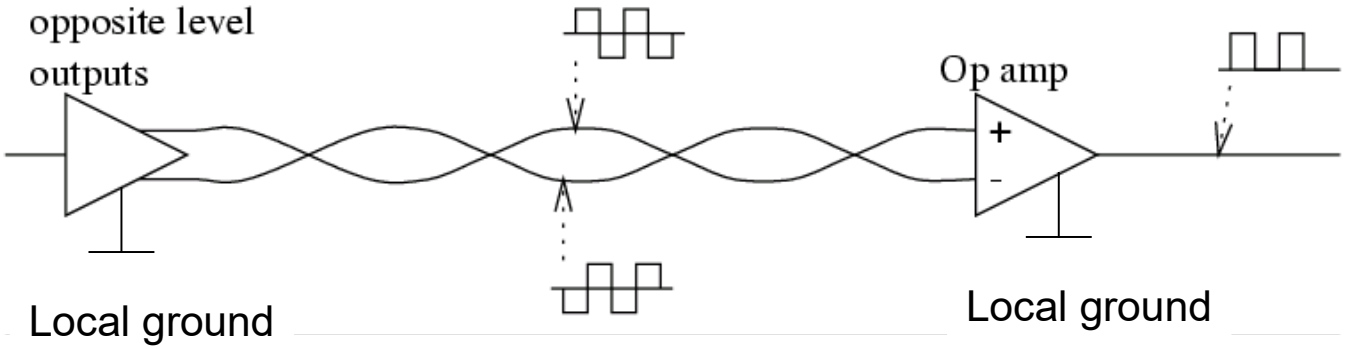
- Single-ended signals

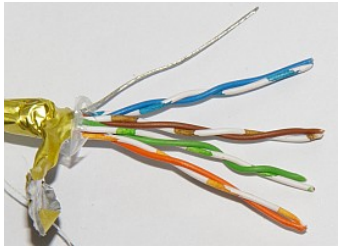


e.g.: RS-232

Voltage at input of Op-Amp positive → '1'; otherwise → '0'

- Differential signals

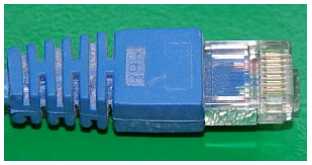




Combined with twisted pairs; Most noise added to both wires.

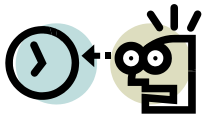
Evaluation: differential Signals

- Advantages
  - Subtraction removes most of the noise
  - Changes of voltage levels have no effect
  - Reduced importance of ground wiring
  - Higher speed
- Disadvantages
  - Requires negative voltages
  - Increased number of wires and connectors
- Applications
  - High-quality analog audio signals (XLR)
  - differential SCSI
  - Ethernet (STP/UTP CAT 5/6/7 cables)
  - FireWire, ISDN, USB



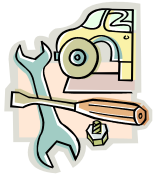
Communication
- Requirements -

- Real-time behavior



- Efficient, economical
  (e.g. centralized power supply)

- Appropriate bandwidth and communication delay
- Robustness
- Fault tolerance
- Diagnosability

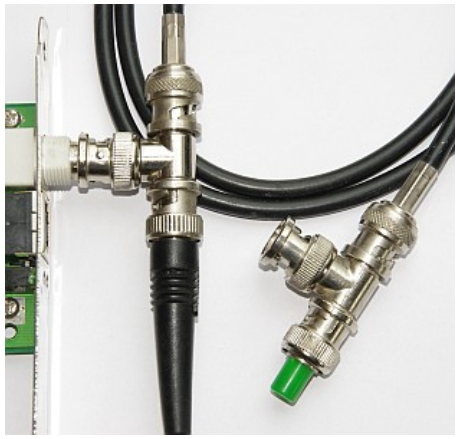


- Maintainability
- Security

- Safety

CSMA/CD vs. CSMA/CA

- Carrier-sense multiple-access/collision-**detection** (CSMA/C**D**, variants of Ethernet)
  - collision ☞ retries
  - no guaranteed response time.

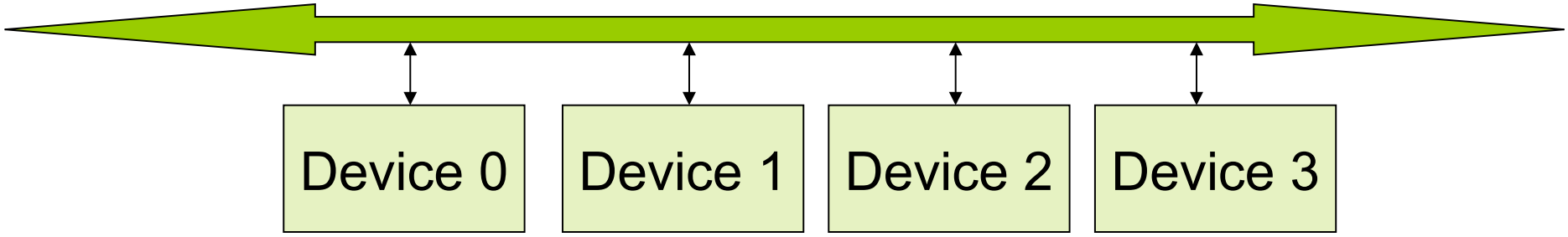


Alternatives

- Carrier-sense multiple-access/ collision-**avoidance** (CSMA/C**A**)
  - WLAN techniques with request preceding transmission
  - Each partner gets an ID (priority).
    After bus transfer: partners try setting their ID;
    Detection of higher ID ☞ disconnect. Guaranteed response time for highest ID, others if given a chance.
- token rings, token busses

Priority-based arbitration of communication media

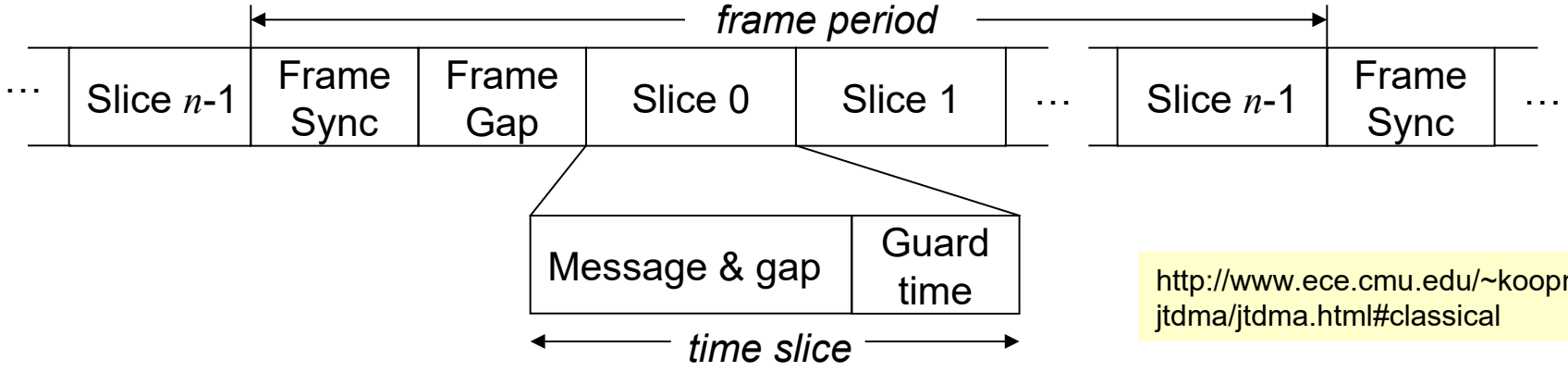
Example: bus



- Bus arbitration (allocation) is frequently priority-based
- ☞ Communication delay depends on communication traffic of other partners
- ☞ No tight real-time guarantees, except for highest priority partner

Time division multiple access (TDMA) busses

Each communication partner is assigned a fixed time slot.
Example:



http://www.ece.cmu.edu/~koopman/jtdma/jtdma.html#classical

E. Wandeler, L. Thiele: Optimal TDMA Time Slot and Cycle Length Allocation for Hard Real-Time Systems, ASP-DAC, 2006

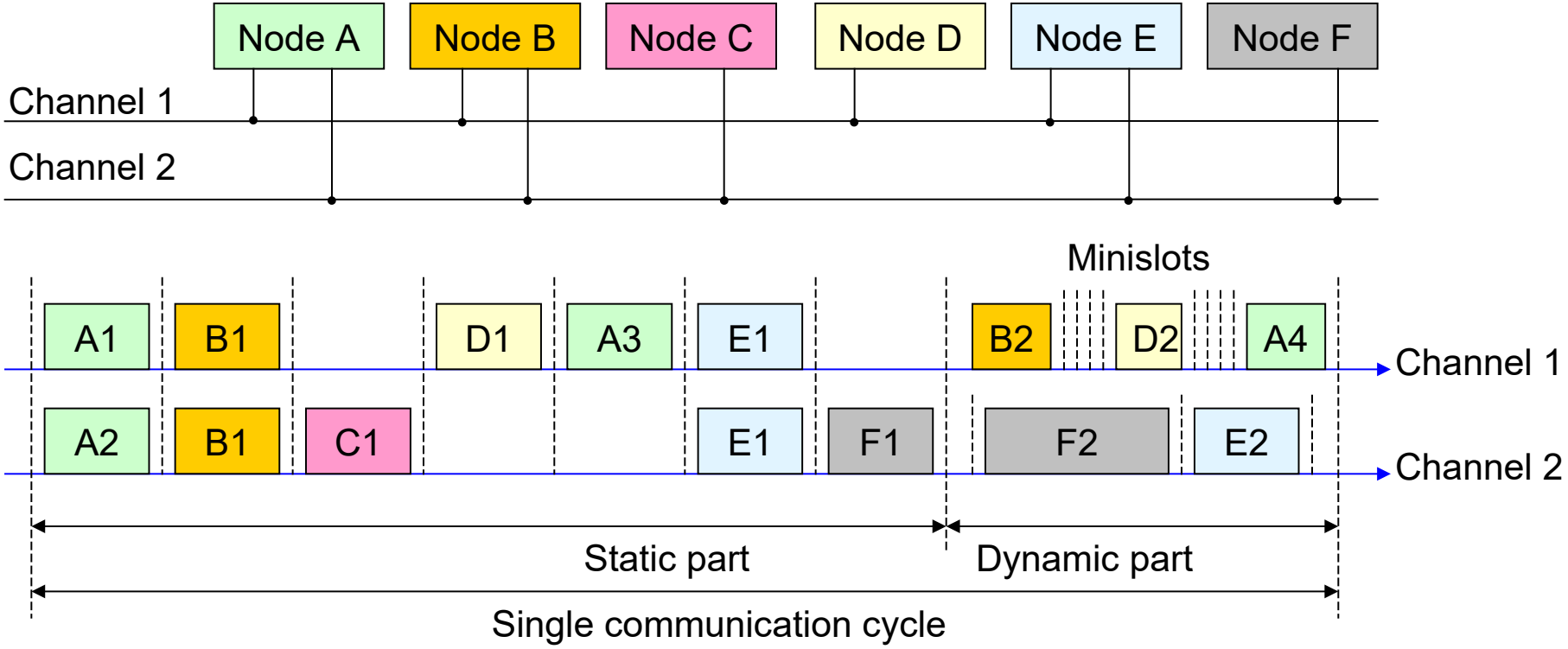
- Master sends sync
- Some waiting time
- Each slave transmits in its time slot
- ∃ variations (truncating unused slots, >1 slots per slave)
- TDMA resources have a deterministic timing behavior
- TDMA provides QoS guarantees in networks on chips

FlexRay

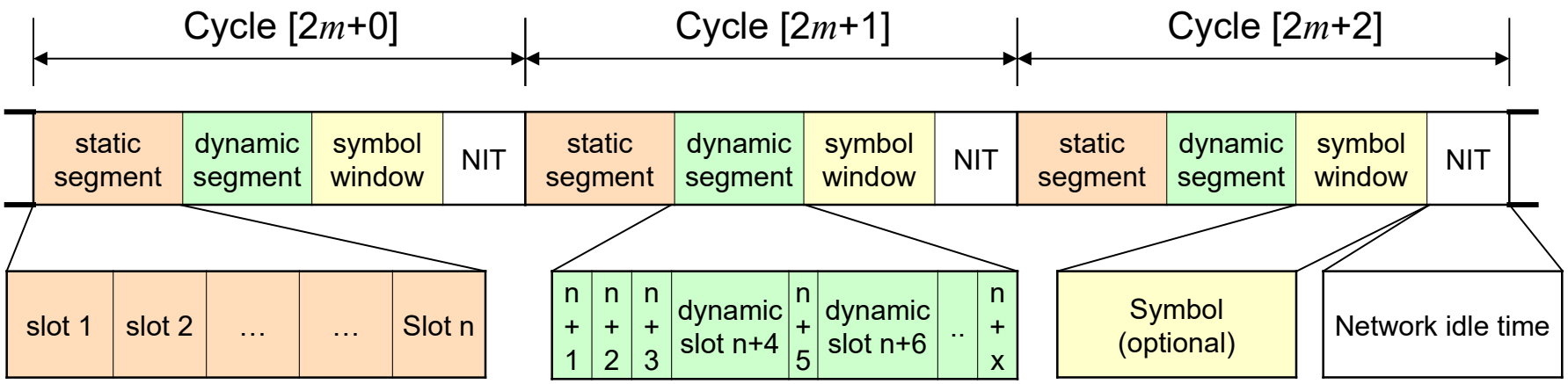
- Developed by the FlexRay consortium (BMW, Ford, Bosch, DaimlerChrysler, …)
- Specified in SDL
- Meets requirements with transfer rates >> CAN standard
  High data rate can be achieved:
  - initially targeted for ~ 10Mbit/sec;
  - design allows much higher data rates
- Improved error tolerance and time-determinism
- TDMA protocol
- Cycle subdivided into a static and a dynamic segment.

TDMA in FlexRay

Exclusive bus access enabled for short time in each case.
Dynamic segment for transmission of variable length information.
Fixed priorities in dynamic segment: Minislots for each potential sender.
Bandwidth used only when it is actually needed.



Time intervals in Flexray

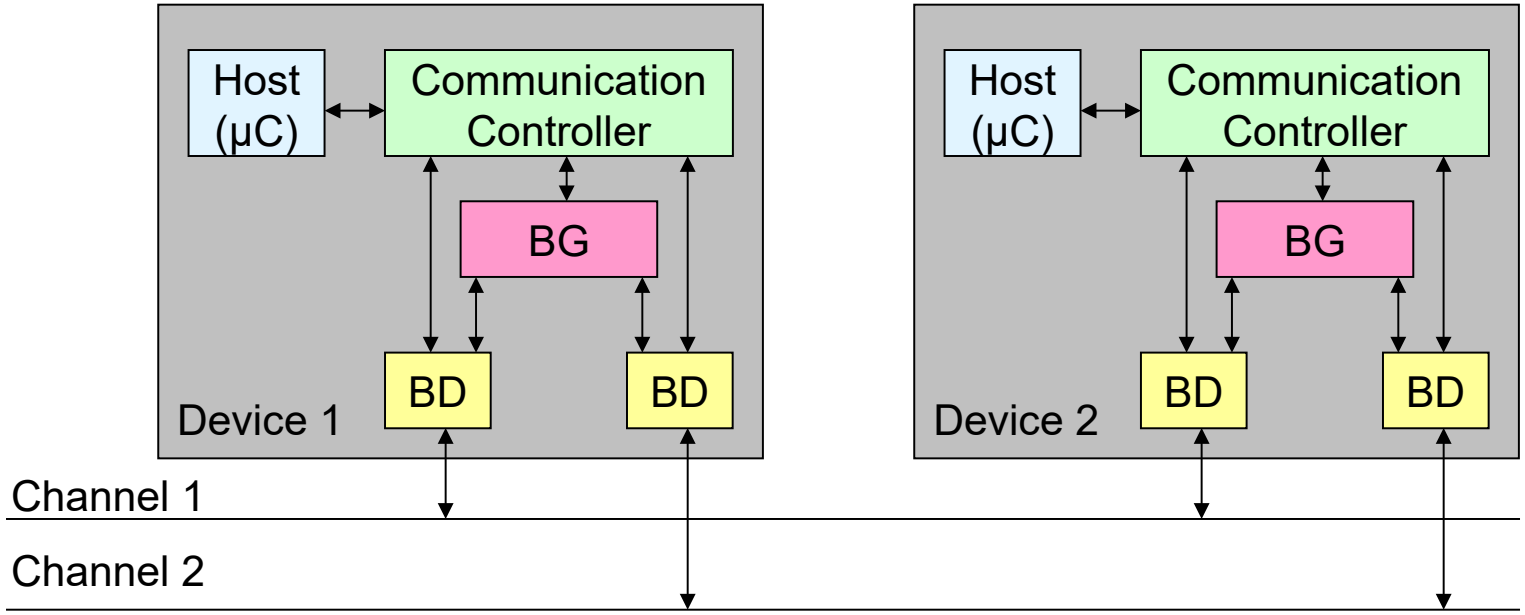


- **Microtick (μt)** = Clock period in partners, may differ between partners
- **Macrotick (mt)** = Basic unit of time, synchronized between partners (=$r_i \times \mu t$, r_i varies between partners i)
- **Slot**=Interval allocated per sender in static segment (=$p \times mt$, p: fixed (configurable))
- **Minislot** = Interval allocated per sender in dynamic segment (=$q \times mt$, q: variable) Short minislot if no transmission needed; starts after previous minislot.
- **Cycle** = static segment + dynamic segment + symbol window/network idle time

Simplified version of diagram by Prof. Form, TU Braunschweig, 2007

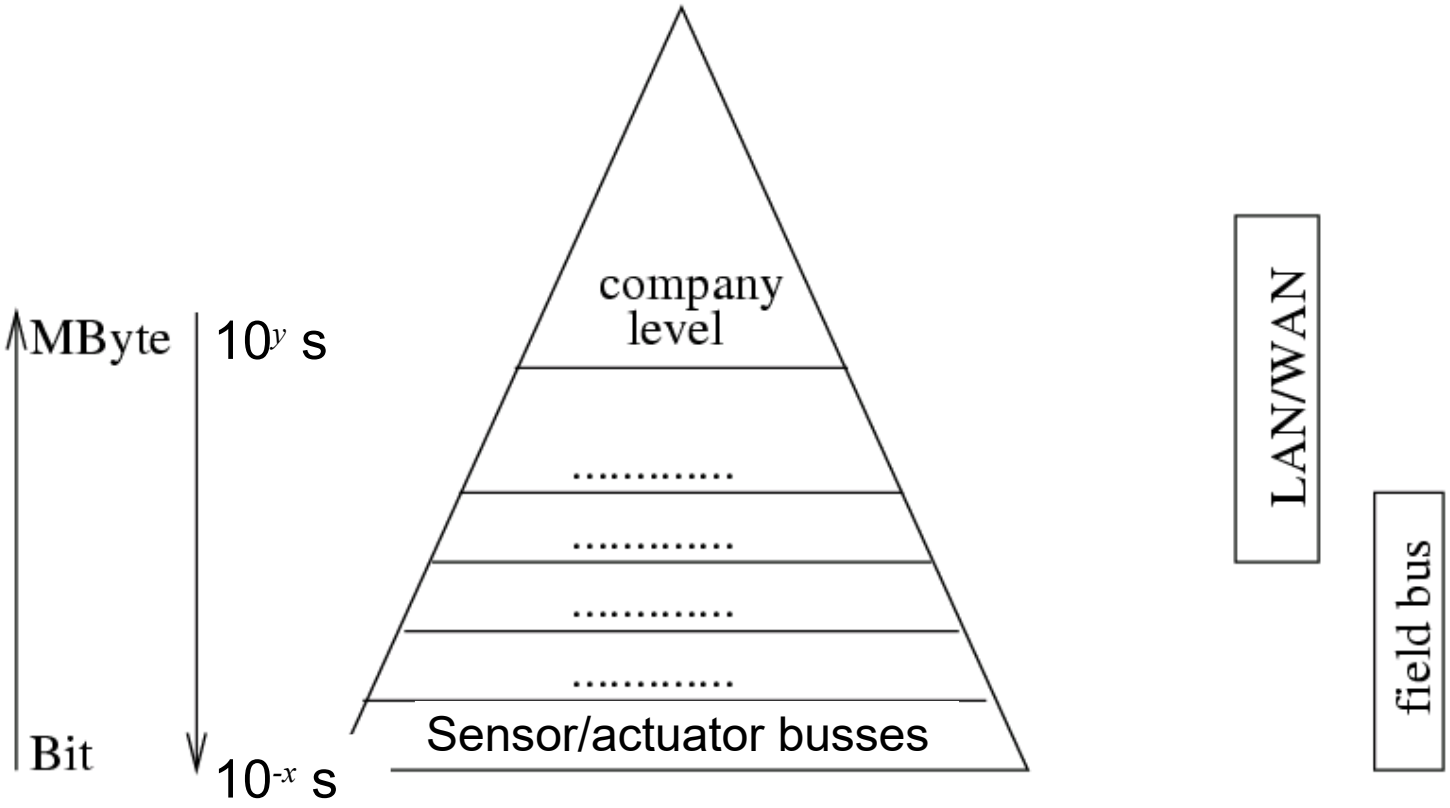
Structure of Flexray networks

Bus guardian BG protects the system against failing processors, e.g. so-called “babbling idiots”



Communication: Hierarchy

Inverse relation between volume and urgency quite common:



Other busses

- **IEEE 488:** Designed for laboratory equipment.
- **CAN:** Controller bus for automotive
- **LIN:** low cost bus for interfacing sensors/actuators in the automotive domain
- **MOST:** Multimedia bus for the automotive domain (not a field bus)
- **MAP:** bus designed for car factories.
- **Process Field Bus (Profibus):** used in smart buildings
- **The European Installation Bus (EIB):** bus designed for smart buildings; CSMA/CA; low data rate.
- Attempts to use Ethernet. Timing predictability an issue.

Wireless communication: Examples

- IEEE 802.11 a/b/g/n
- UMTS; HSPA; LTE
- DECT
- Bluetooth
- ZigBee

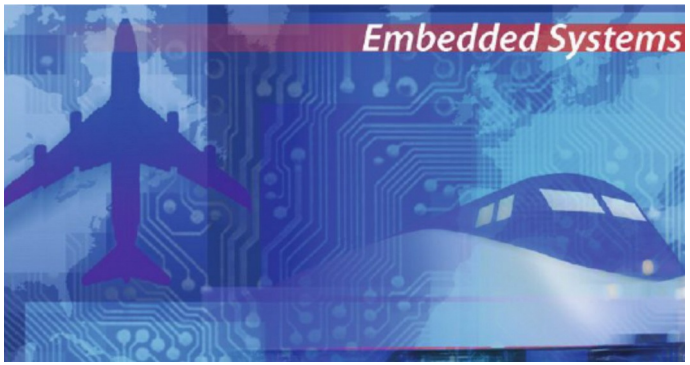
Timing predictability of wireless communication?

Energy consumption e.g. of Wimax devices?

☞ chapter 5

D/A-Converters

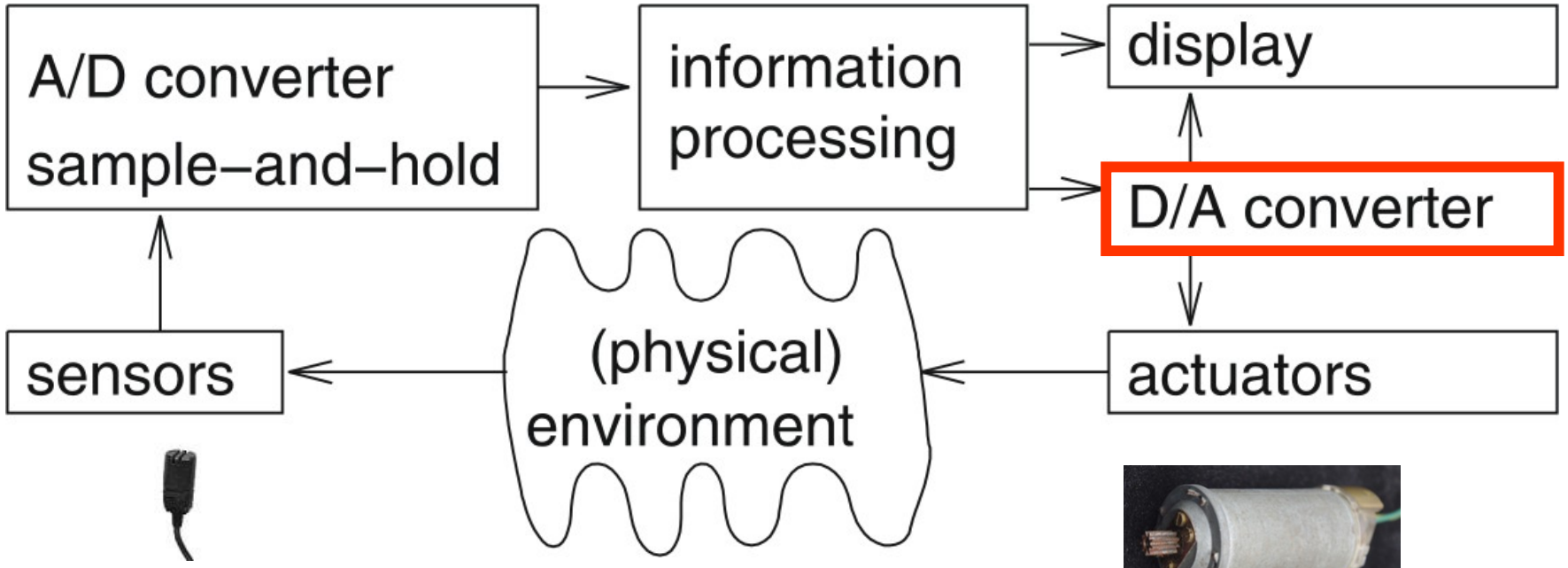
Peter Marwedel
TU Dortmund,
Informatik 12



2012 年 11 月 21 日

Embedded System Hardware

Embedded system hardware is frequently used in a loop (***"hardware in a loop"***):

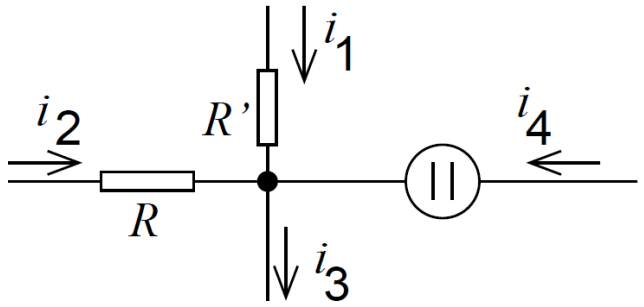


☞ cyber-physical systems

Kirchhoff´s junction rule
Kirchhoff´s Current Law, Kirchhoff´s first rule

Kirchhoff´s Current Law:
At any point in an electrical circuit, the sum of currents flowing towards that point is equal to the sum of currents flowing away from that point.
(Principle of conservation of electric charge)

Example:



$$i_1 + i_2 + i_4 = i_3$$

$$i_1 + i_2 - i_3 + i_4 = 0$$

Formally, for any node in a circuit:

$$\sum_k i_k = 0$$

Count current flowing away from node as negative.

Kirchhoff´s loop rule
Kirchhoff´s Voltage Law, Kirchhoff´s second rule

The principle of conservation of energy implies that:

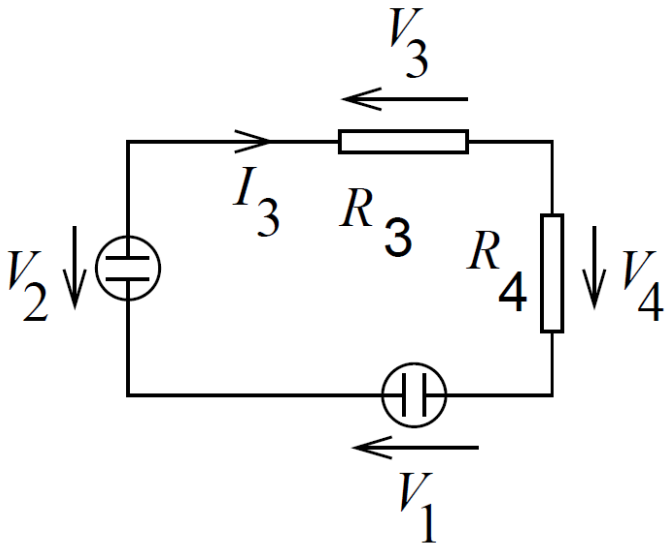
The sum of the potential differences (voltages) across all elements around any closed circuit must be zero

Formally, for any loop in a circuit:

$$\sum_k V_k = 0$$

Count voltages traversed against arrow direction as negative

Example:



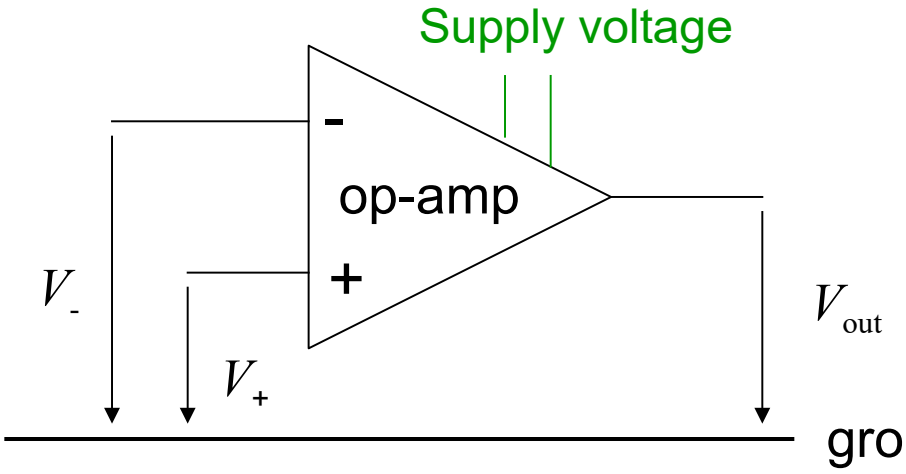
$$V_1 - V_2 - V_3 + V_4 = 0$$

$V_3 = R_3 \times I_3$ if current counted in the same direction as V_3

$V_3 = -R_3 \times I_3$ if current counted in the opposite direction as V_3

Operational Amplifiers (Op-Amps)

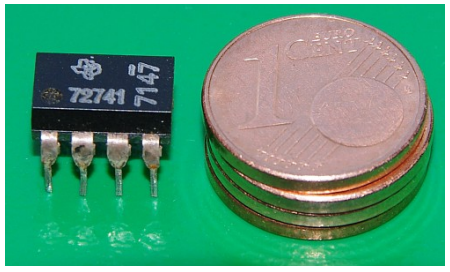
Operational amplifiers (op-amps) are devices amplifying the voltage difference between two input terminals by a large gain factor g



$$V_{out}=(V_+ - V_-) \cdot g$$

High impedance input terminals
$\Rightarrow$ Currents into inputs ≈ 0

Op-amp in a DIL package

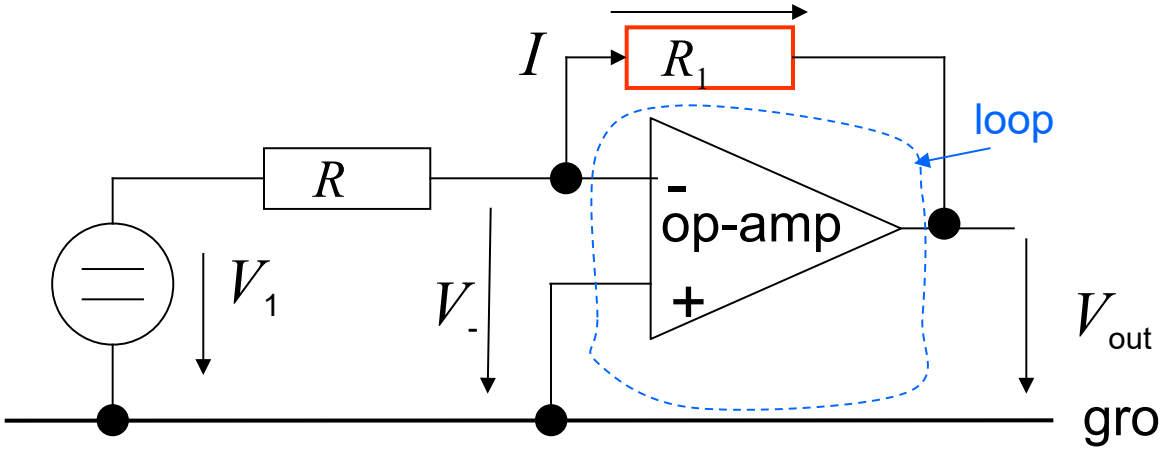


For an **ideal** op-amp: $g \rightarrow \infty$

(In practice: g may be around $10^4..10^6$)

Op-Amps with feedback

In circuits, negative feedback is used to define the actual gain:



Due to the feedback to the *inverted* input, R_1 reduces voltage V_-. To which level?

$V_{out} = -g \cdot V_-$ (op-amp feature)

$I \cdot R_1 + V_{out} - V_- = 0$ (loop rule)

$\Rightarrow I \cdot R_1 + -g \cdot V_- - V_- = 0$

$\Rightarrow (1+g) \cdot V_- = I \cdot R_1$

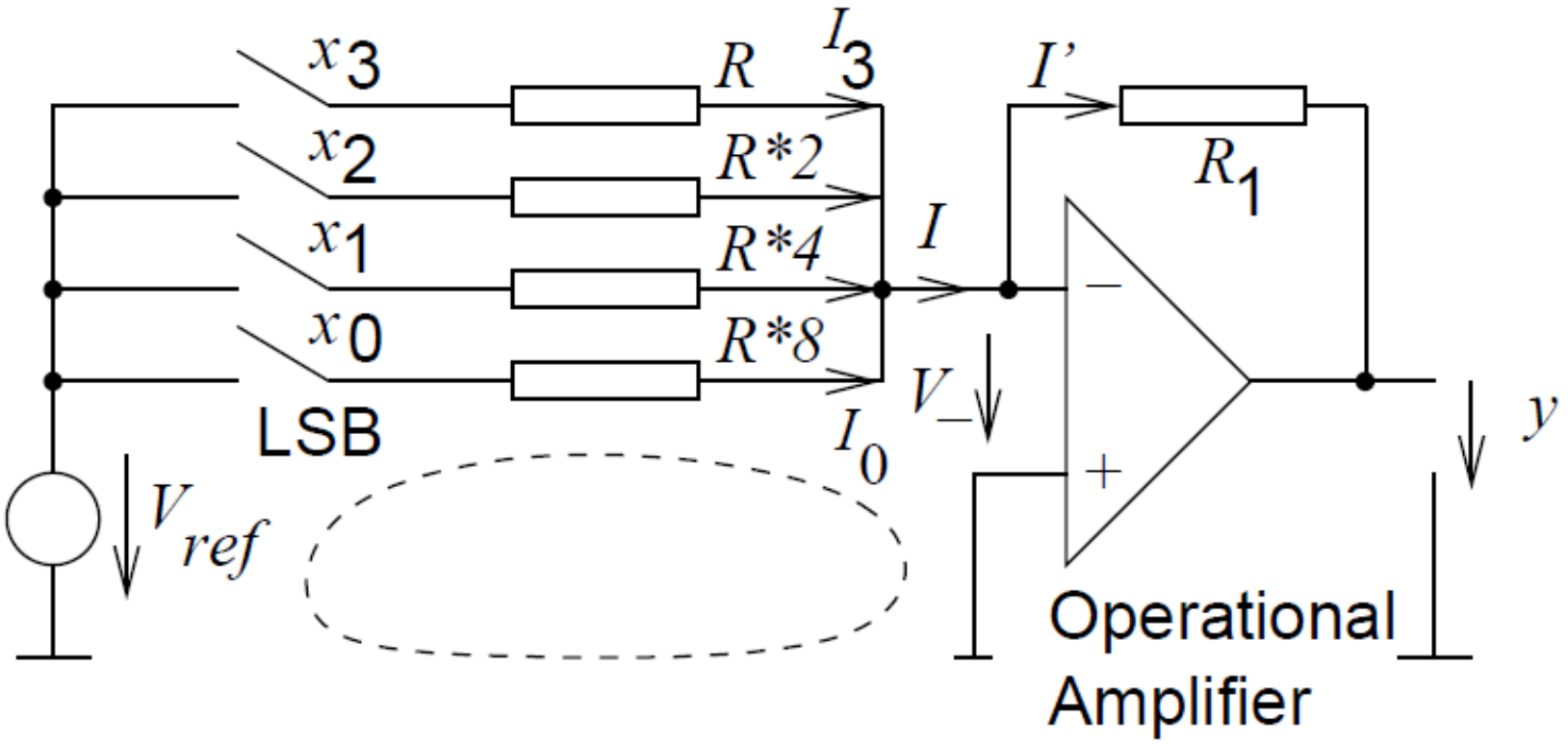
$$\Rightarrow V_- = \frac{I \cdot R_1}{1+g}$$

$$V_{-,ideal} = \lim_{g \to \infty} \frac{I \cdot R_1}{1+g} = 0$$

V_- is called **virtual ground**: the voltage is 0, but the terminal may not be connected to ground

Digital-to-Analog (D/A) Converters

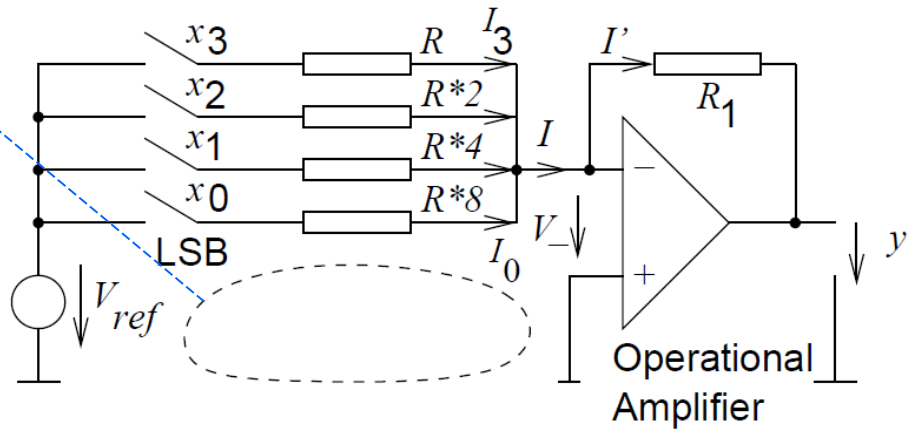
Various types, can be quite simple,
e.g.:



Current I proportional to the number represented by x

Loop rule:

$$x_0 \cdot I_0 \cdot 8 \cdot R + V_- - V_{ref} = 0$$



$$\Rightarrow \quad I_0 = x_0 \times \frac{V_{ref}}{8 \times R}$$

In general: $$I_i = x_i \times \frac{V_{ref}}{2^{3-i} \times R}$$

Junction rule: $$I = \sum_i I_i$$

$$\Rightarrow \quad I = x_3 \times \frac{V_{ref}}{R} + x_2 \times \frac{V_{ref}}{2 \times R} + x_1 \times \frac{V_{ref}}{4 \times R} + x_0 \times \frac{V_{ref}}{8 \times R} = \frac{V_{ref}}{8 \times R} \times \sum_{i=0}^{3} x_i \times 2^i$$

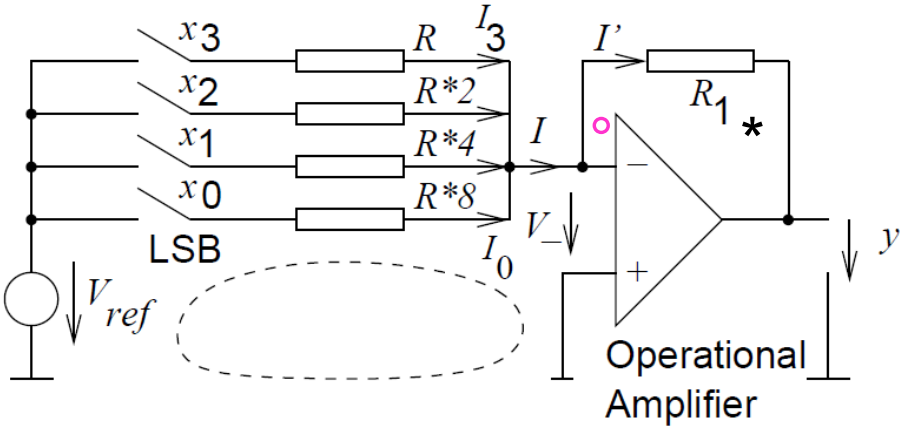
$I \sim nat\,(x)$, where $nat(x)$: natural number represented by x;

Output voltage proportional to the number represented by x

Loop rule*: $y + R_1 \times I' = 0$

Junction rule°: $I = I'$

☞ $y + R_1 \times I = 0$



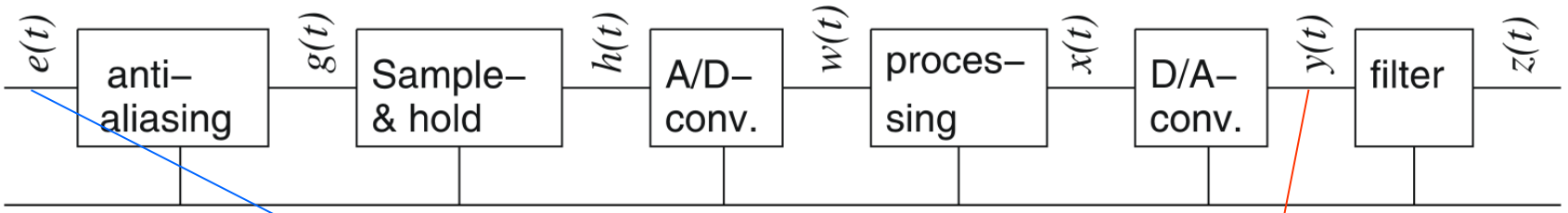
From the previous slide

$$I = \frac{V_{ref}}{8 \times R} \times \sum_{i=0}^{3} x_i \times 2^i$$

Hence: $$y = -V_{ref} \times \frac{R_1}{8 \times R} \sum_{i=0}^{3} x_i \times 2^i = -V_{ref} \times \frac{R_1}{8 \times R} \times nat(x)$$

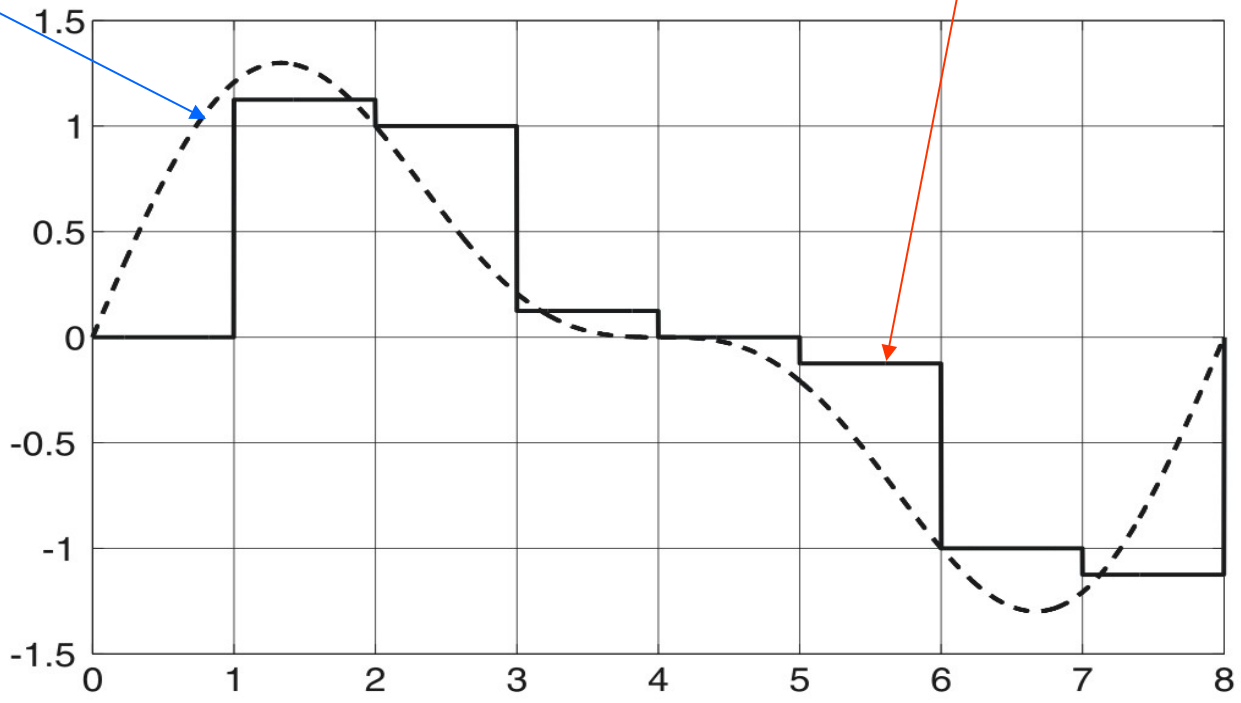
Op-amp turns current $I \sim nat\,(x)$ into a voltage $\sim nat\,(x)$

Output generated from signal $e_3(t)$



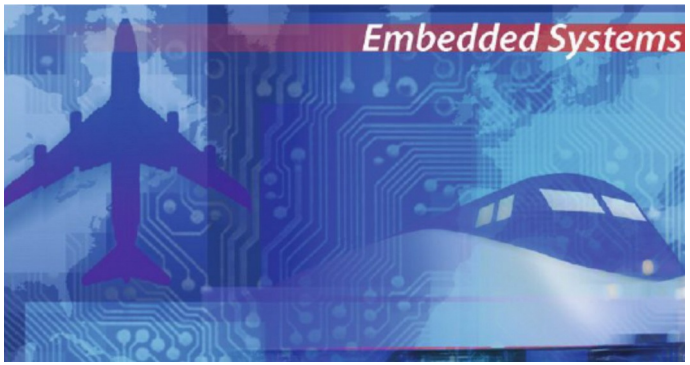
* Assuming "zero-order hold"

Possible to reconstruct input signal?



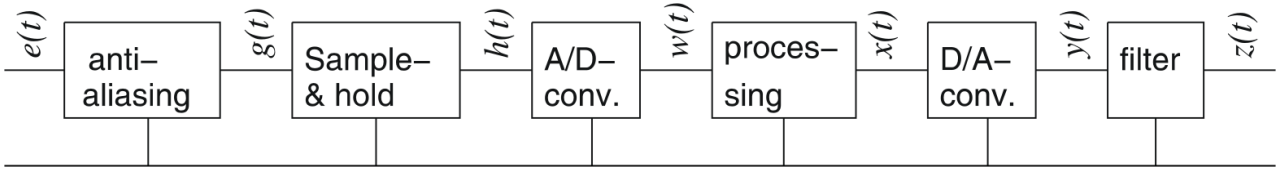
Sampling Theorem

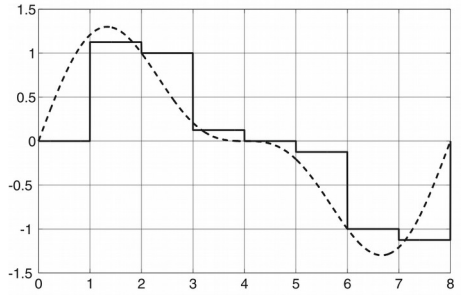
Peter Marwedel
TU Dortmund,
Informatik 12



2012 年 11 月 21 日

Possible to reconstruct input signal?





- Assuming Nyquist criterion met
- Let $\{t_s\}$, $s = ...,-1,0,1,2, ...$ be times at which we sample $g(t)$
- Assume a constant sampling rate of $1/p_s$ ($\forall s$: $p_s = t_{s+1} - t_s$).
- According sampling theory, we can approximate the input signal as follows:

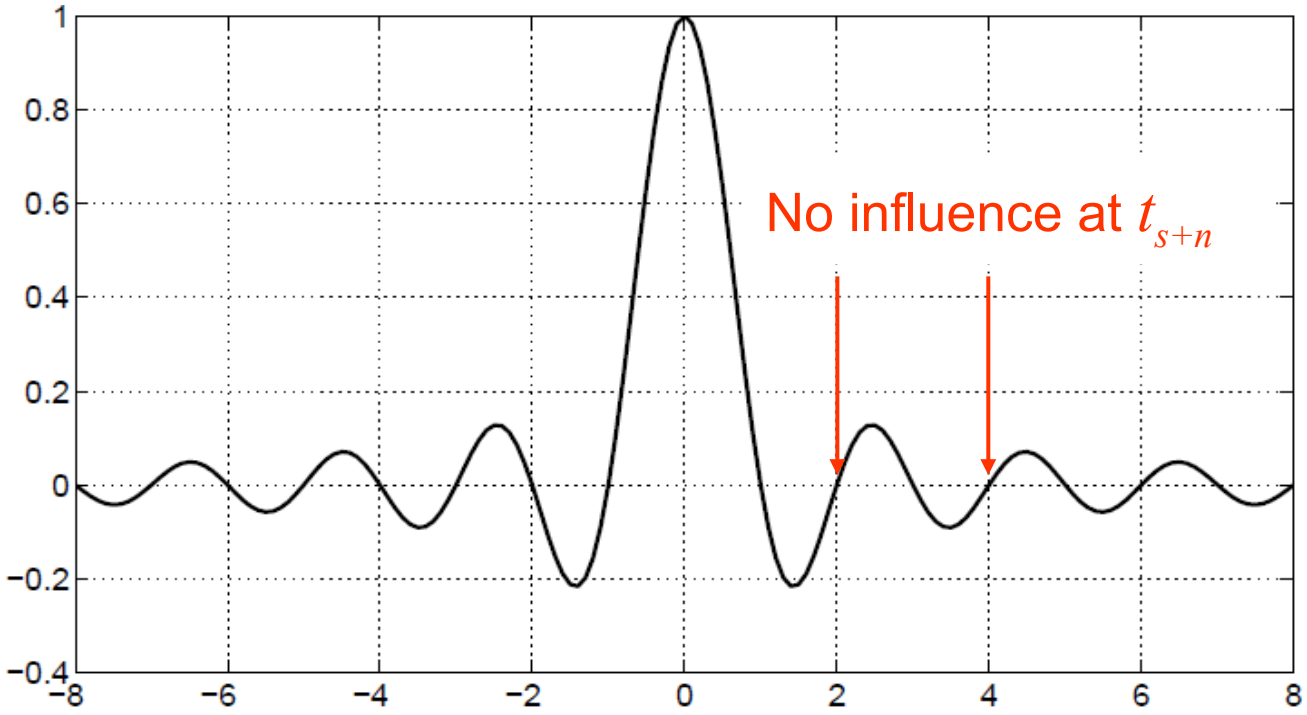
$$z(t) = \sum_{s=-\infty}^{\infty} \frac{y(t_s)\sin\frac{\pi}{p_s}(t-t_s)}{\frac{\pi}{p_s}(t-t_s)}$$

Weighting factor for influence of $y(t_s)$ at time t

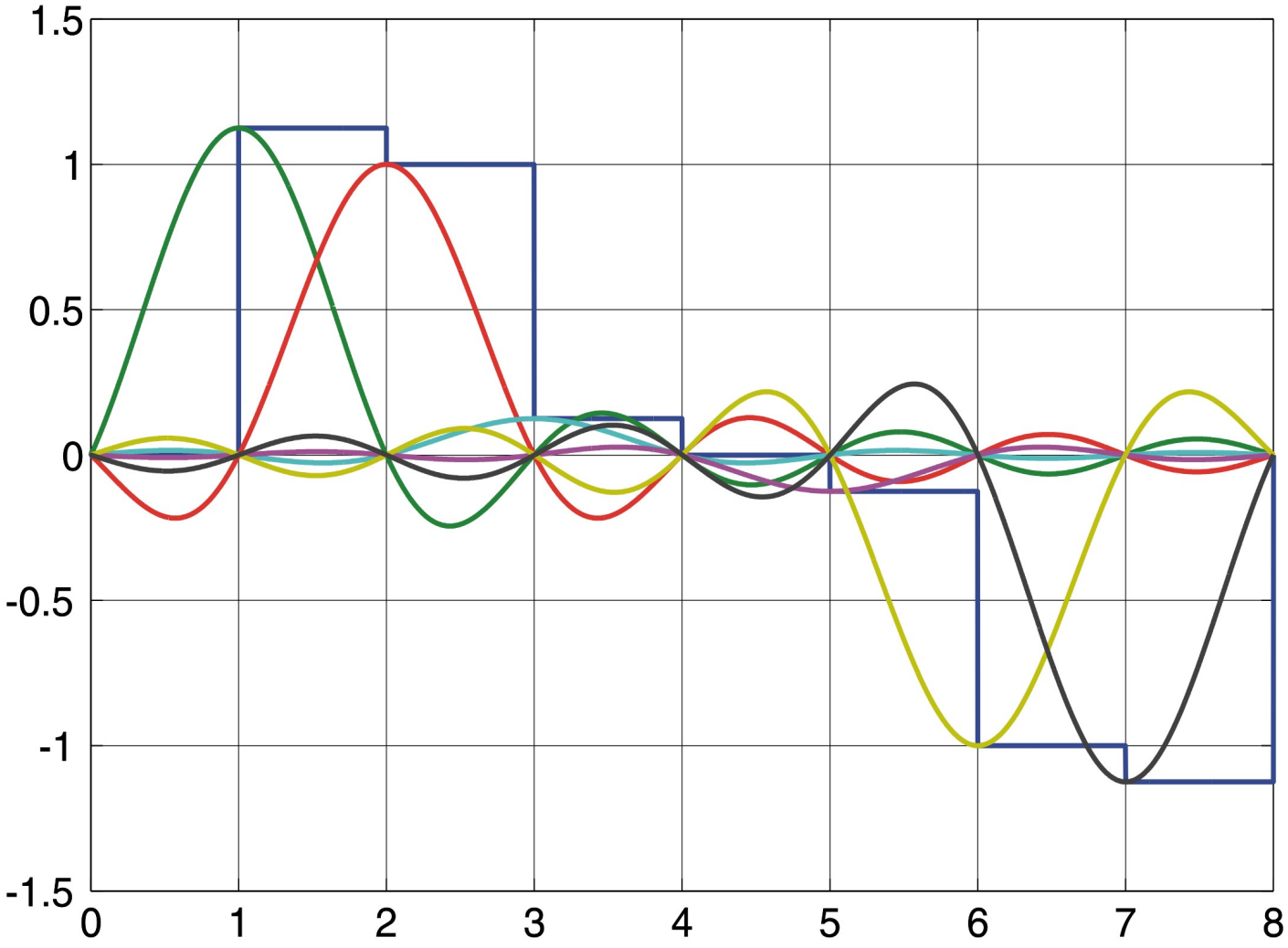
Oppenheim, Schafer, 2009

Weighting factor for influence of $y(t_s)$ at time t

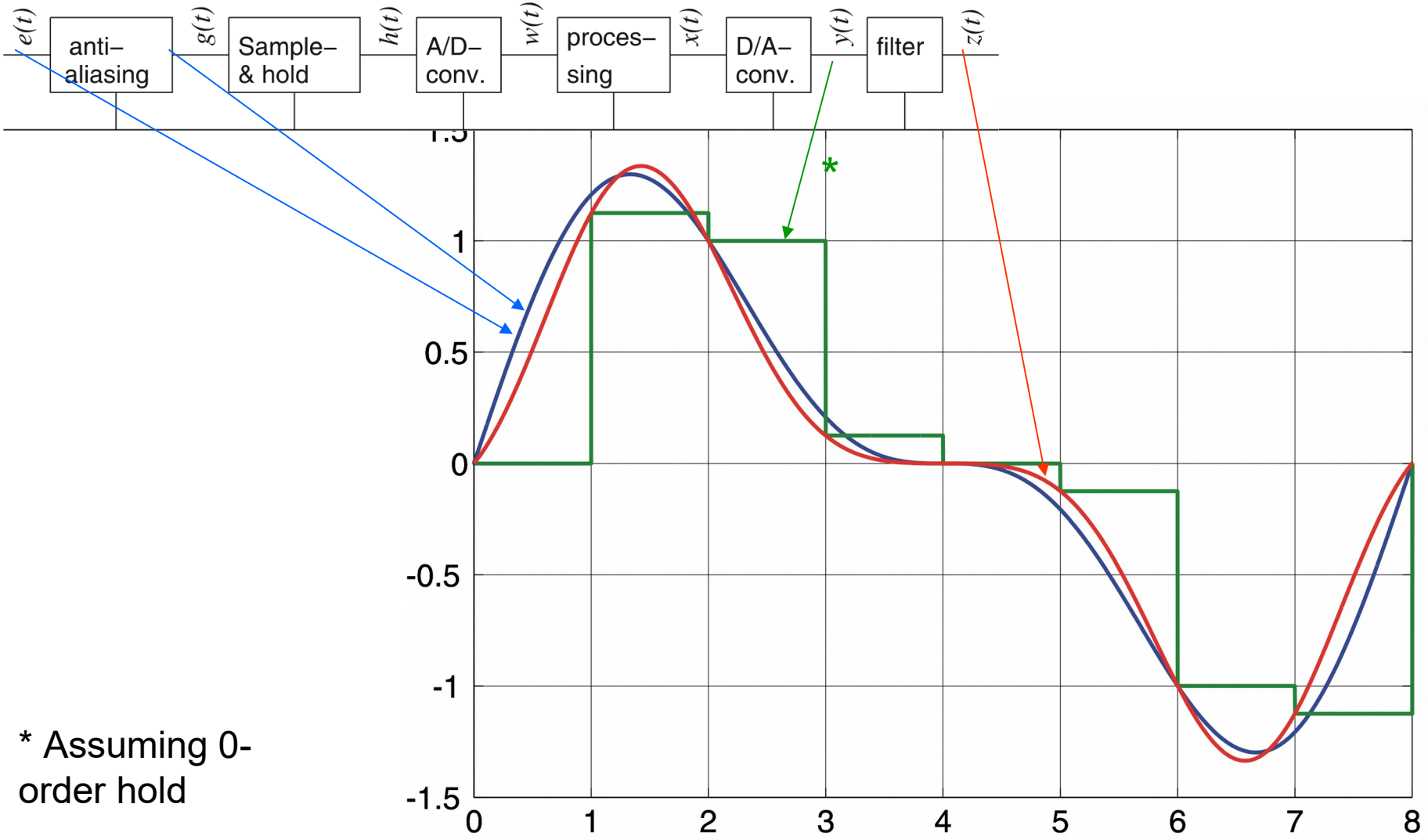
$$sinc(t-t_s)=\frac{\sin\left(\frac{\pi}{p_s}(t-t_s)\right)}{\frac{\pi}{p_s}(t-t_s)}$$



Contributions from the various sampling instances



(Attempted) reconstruction of input signal

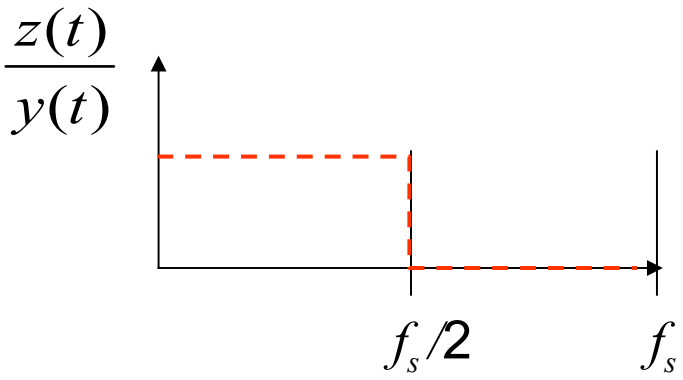


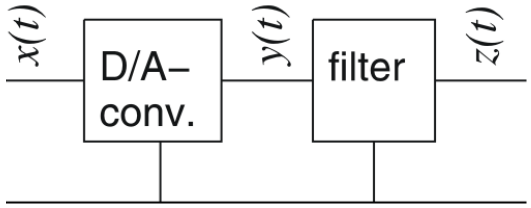
* Assuming 0-order hold

How to compute the *sinc*( ) function?

$$z(t) = \sum_{s=-\infty}^{\infty} \frac{y(t_s)\sin\frac{\pi}{p_s}(t-t_s)}{\frac{\pi}{p_s}(t-t_s)}$$

- **Filter theory:** The required interpolation is performed by an ideal low-pass filter (*sinc* is the Fourier transform of the low-pass filter transfer function)





Filter removes high frequencies present in $y(t)$

How precisely are we reconstructing the input?

$$z(t) = \sum_{s=-\infty}^{\infty} \frac{y(t_s)\sin\frac{\pi}{p_s}(t-t_s)}{\frac{\pi}{p_s}(t-t_s)}$$

- **Sampling theory**:
  - **Reconstruction using *sinc* () is precise**

- However, it may be impossible to really compute $z(t)$ as indicated …

Limitations

$$z(t) = \sum_{s=-\infty}^{\infty} \frac{y(t_s)\sin\frac{\pi}{p_s}(t-t_s)}{\frac{\pi}{p_s}(t-t_s)}$$

- Actual filters do not compute $sinc(\ )$
  In practice, filters are used as an approximation.
  Computing good filters is an art itself!
- All samples must be known to reconstruct $e(t)$ or $g(t)$.
  ☞ Waiting indefinitely before we can generate output!
  In practice, only a finite set of samples is available.
- Actual signals are never perfectly bandwidth limited.
- Quantization noise cannot be removed.

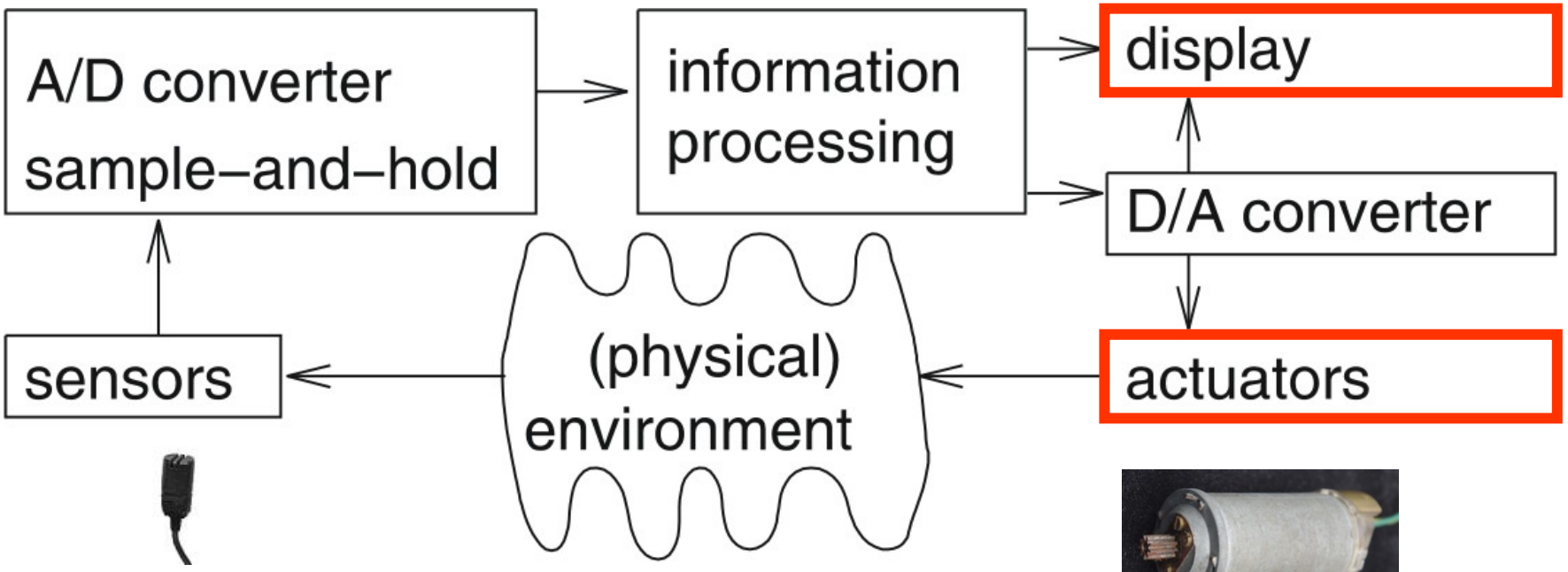
Output

Output devices of embedded systems include

- **Displays:** Display technology is extremely important. Major research and development efforts
- **Electro-mechanical devices:** these influence the environment through motors and other electro-mechanical equipment. Frequently require analog output.

Embedded System Hardware

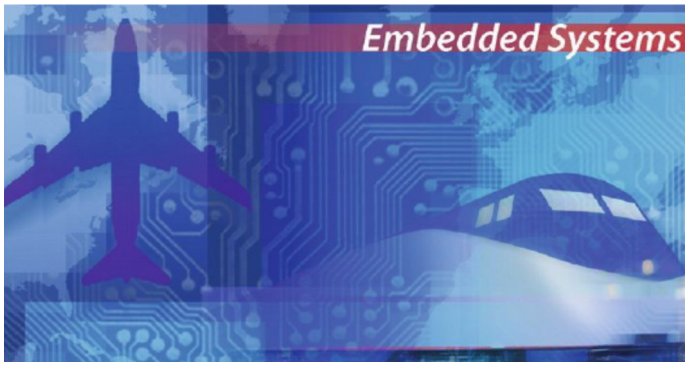
Embedded system hardware is frequently used in a loop (***"hardware in a loop"***):



☞ cyber-physical systems

Actuators

Peter Marwedel
TU Dortmund,
Informatik 12



2012 年 11 月 21 日

Actuators (1)

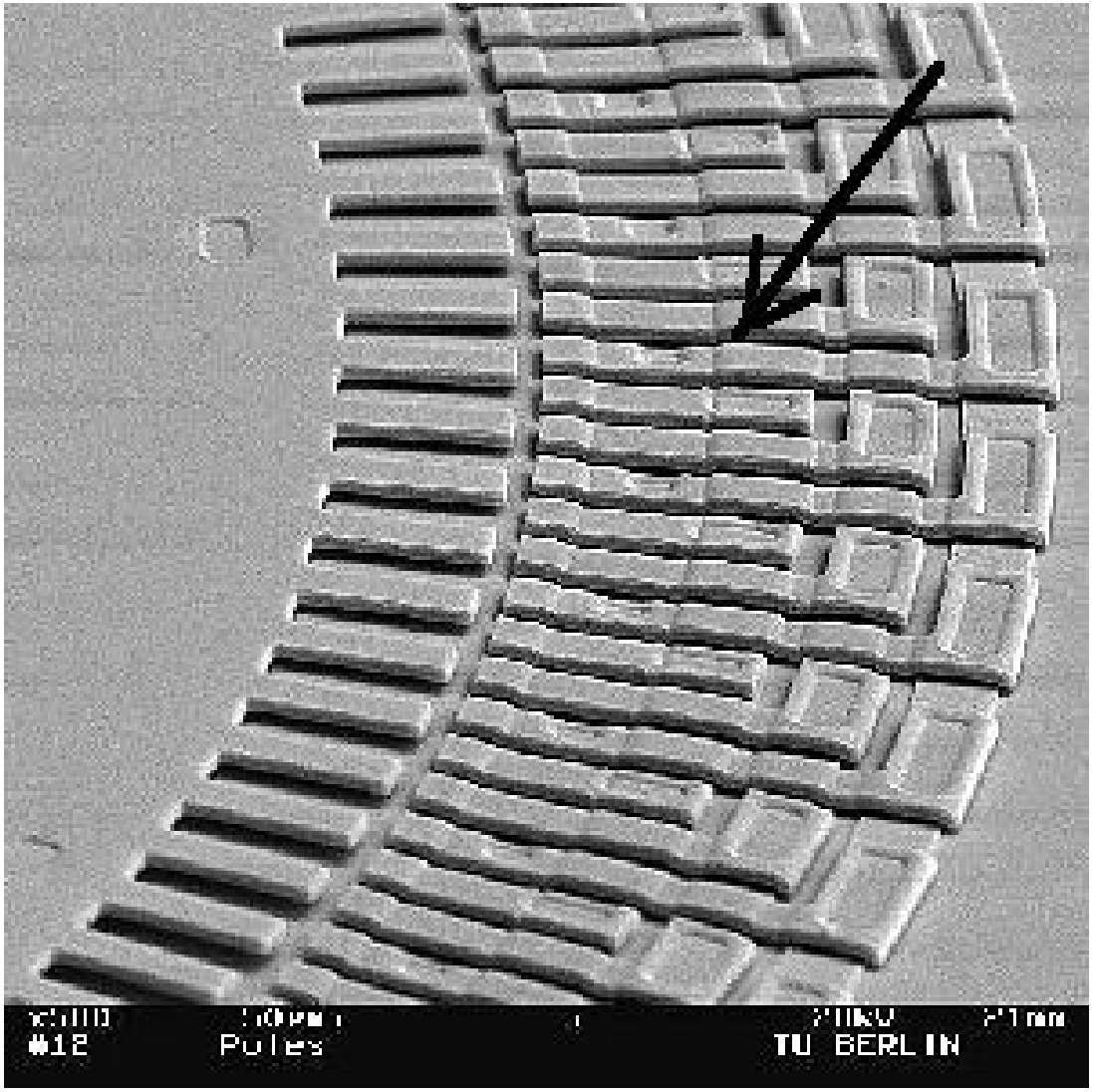
Huge variety of actuators and output devices, impossible to present all of them.

Motor as an example

- electric motor

- piezoelectric
  https://piezomotor.com/technology
- micromechanic

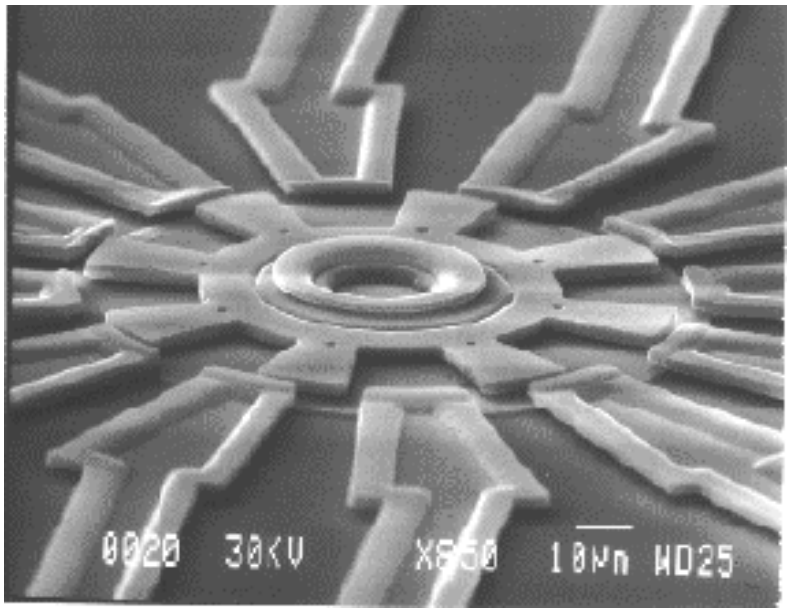
Actuators (2)

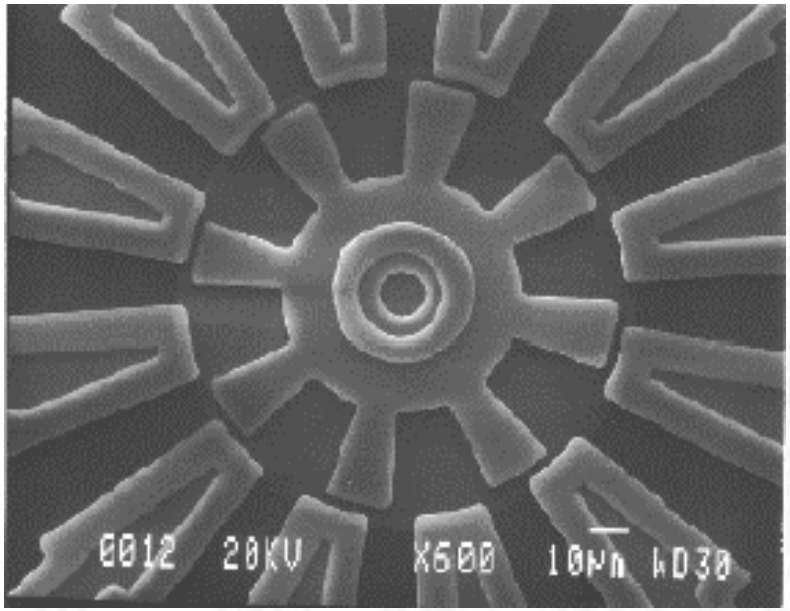


Courtesy and ©:
E. Obermeier, MAT, TU Berlin

OMM-Beispiele (cont.)

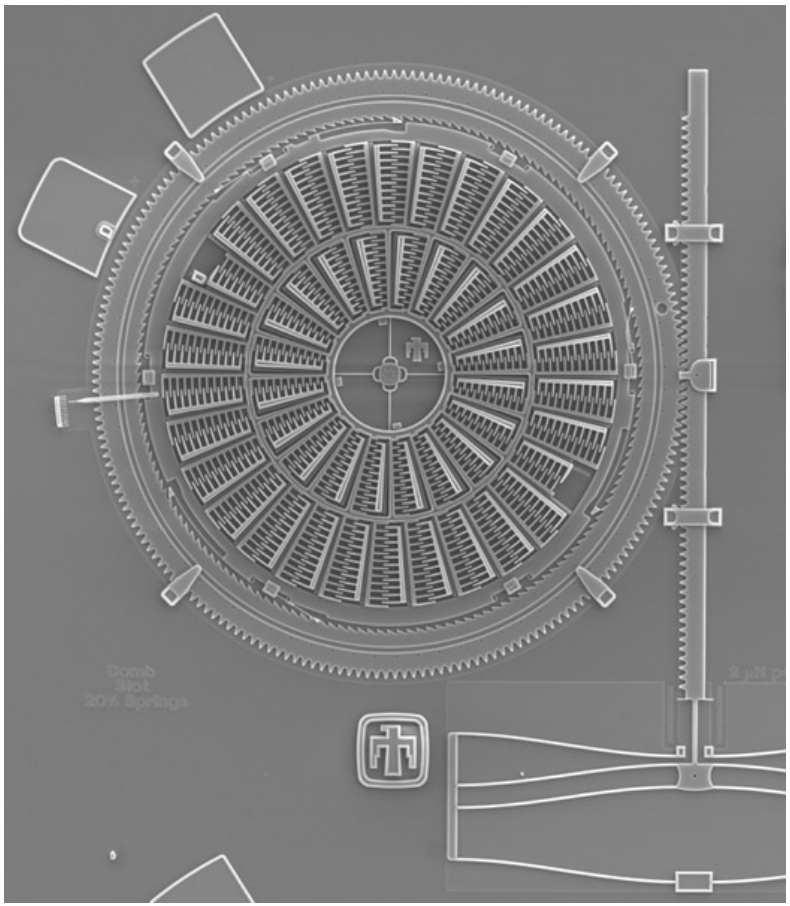
Elektrostatisch angetriebener Motor

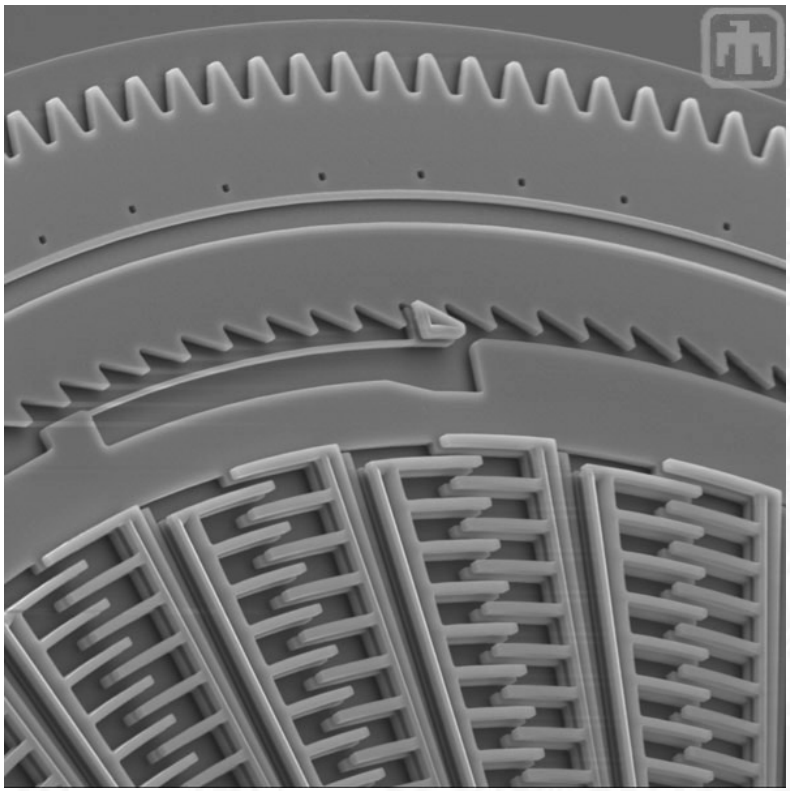




OMM-Beispiele

Elektrostatisch oszillierender Motor





Secure Hardware

- Security needed for communication & storage
- Demand for special equipment for cryptographic keys
- To resist side-channel attacks like
  - measurements of the supply current or
  - Electromagnetic radiation.

  Special mechanisms for physical protection (shielding, sensor detecting tampering with the modules).
- Logical security, using cryptographic methods needed.
- Smart cards: special case of secure hardware
  - Have to run with a very small amount of energy.
- In general, we have to distinguish between different levels of security and knowledge of “adversaries”

Summary

Hardware in a loop

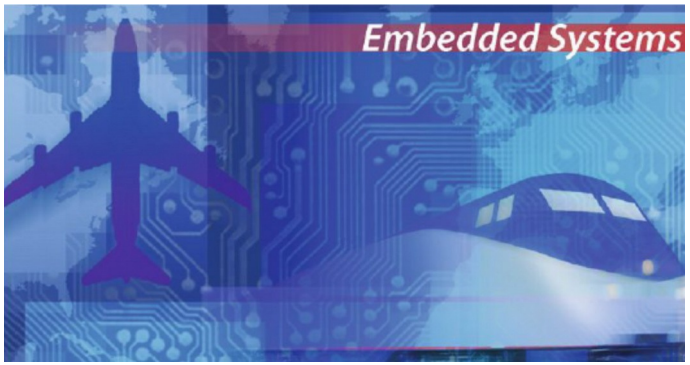
- Sensors
- Discretization
- Information processing
  - Importance of energy efficiency, Special purpose HW very expensive, Energy efficiency of processors, Code size efficiency, Run-time efficiency
  - Reconfigurable Hardware
- Communication
- D/A converters
- Sampling theorem
- Actuators (briefly)
- Secure hardware (briefly)

technische universität
dortmund

fakultät für informatik
informatik 12

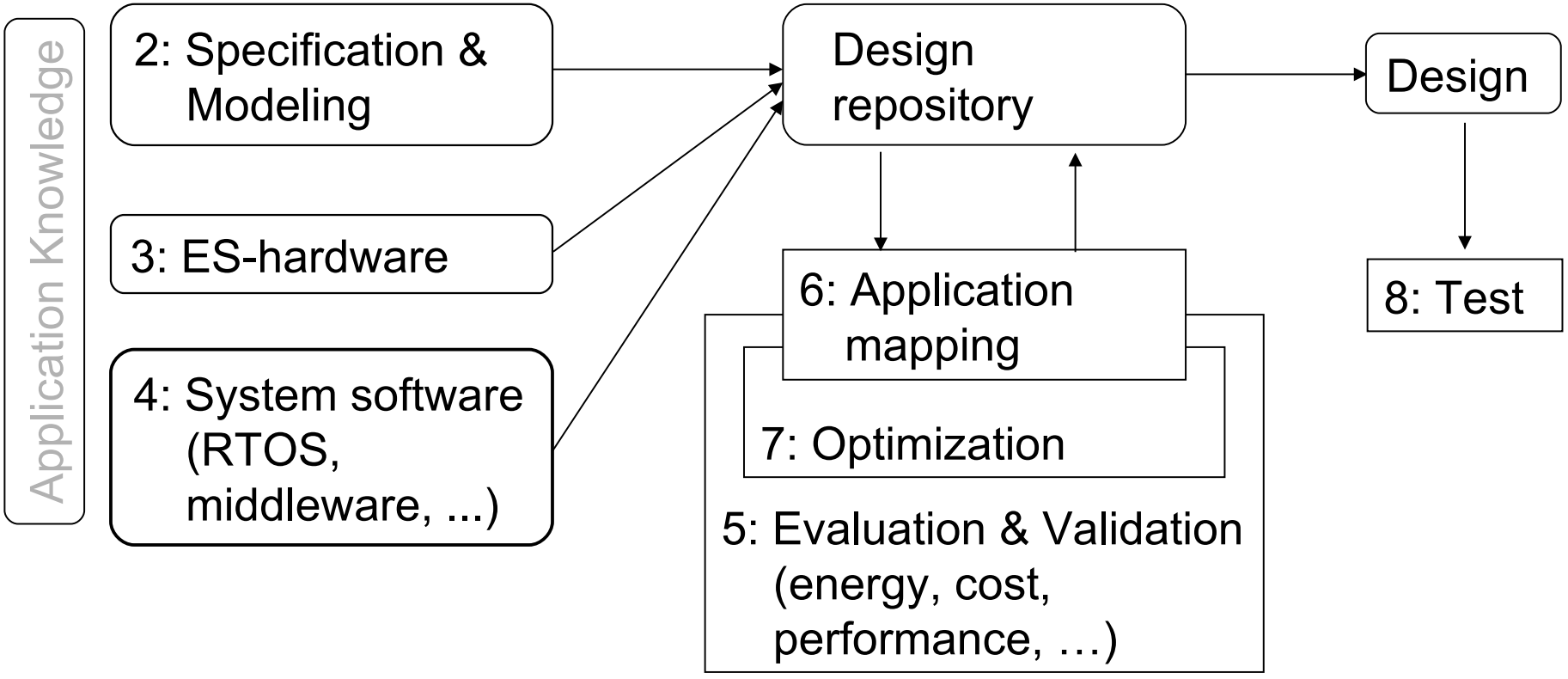
Embedded & Real-time Operating Systems

Peter Marwedel
TU Dortmund,
Informatik 12



2013 年 11 月 26 日

Structure of this course



Numbers denote sequence of chapters

Increasing design complexity + Stringent time-to-market requirements ☞Reuse of components

Reuse requires knowledge from previous designs to be made available in the form of **intellectual property** (IP, for **SW & HW**).

- HW
- Operating systems
- Middleware (Communication, data bases, …)
- ….

Embedded operating systems
- Characteristics: Configurability -

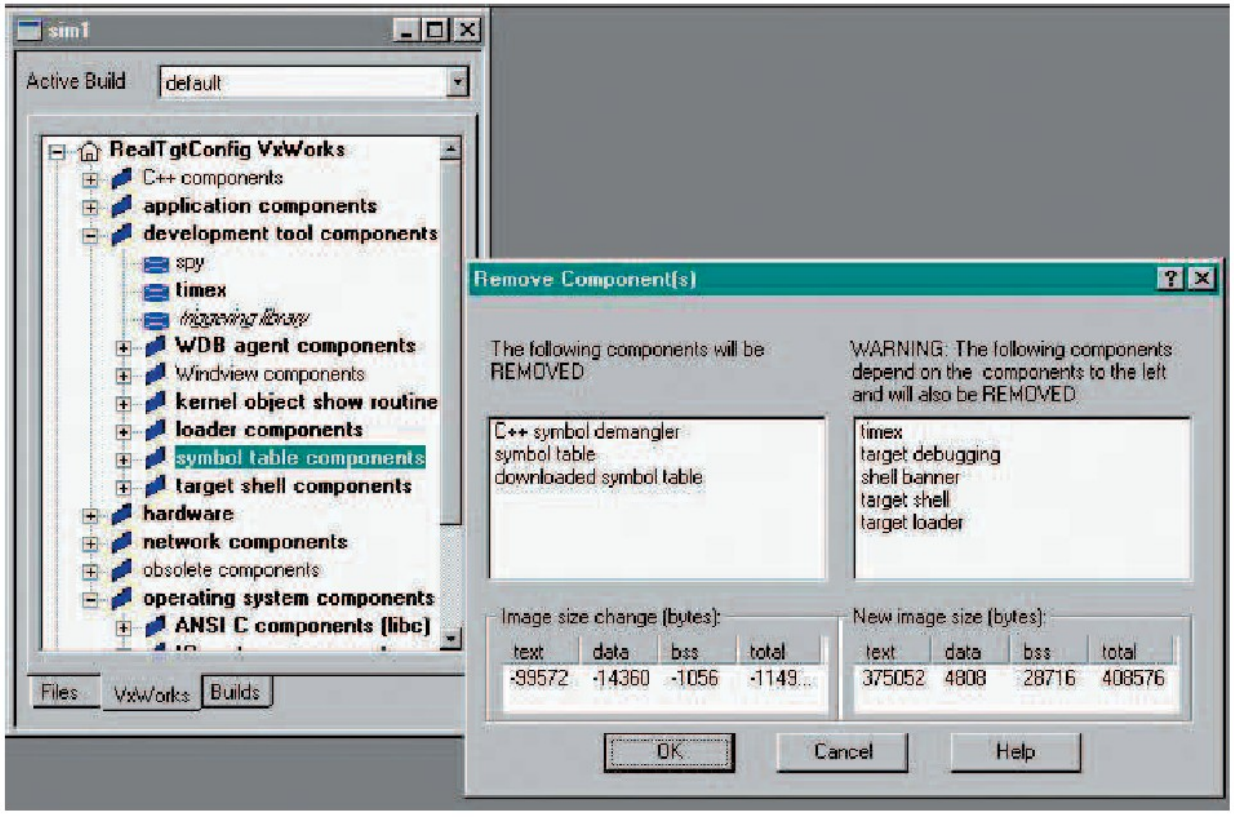
Configurability

No overhead for unused functions tolerated,
no single OS fits all needs, ☞ configurability needed.

- Object-orientation could lead to a of derivation subclasses.
- Aspect-oriented programming
- Conditional compilation (using #if and #ifdef commands).
- Advanced compile-time evaluation useful.
- Linker-time optimization (removal of unused functions)

Dynamic data might be replaced by static data.

Example: Configuration of VxWorks



https://www.windriver.com/products/vxworks

Automatic dependency analysis and size calculations allow users to quickly custom-tailor the VxWORKS operating system.

Verification of derived OS?

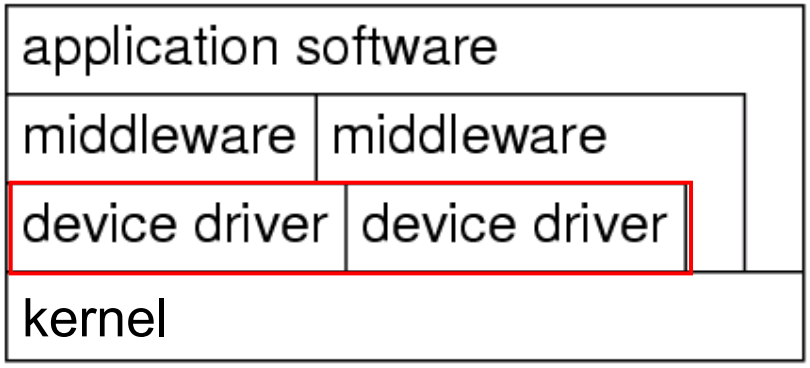
Verification a potential problem of systems with a large number of derived OSs:

- Each derived OS must be tested thoroughly;
- Potential problem for eCos (open source RTOS from Red Hat), including 100 to 200 configuration points [Takada, 2001].

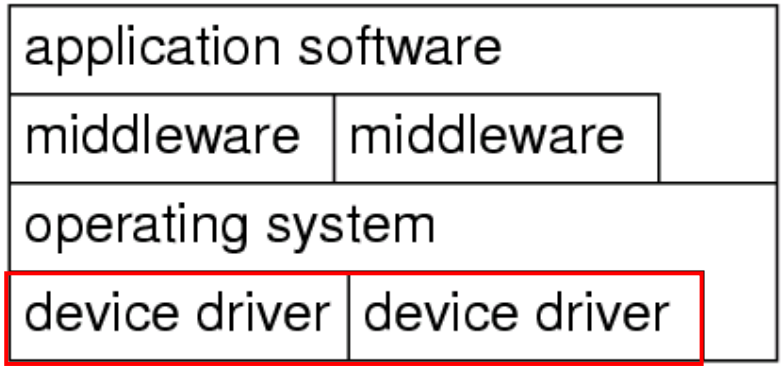
Embedded operating systems
- Characteristics: Devices handled by tasks -

- **Effectively no device needs to be supported by all variants of the OS**, except maybe the system timer.
- Many ES without disk, a keyboard, a screen or a mouse.
- Disk & network handled by tasks instead of integrated drivers.

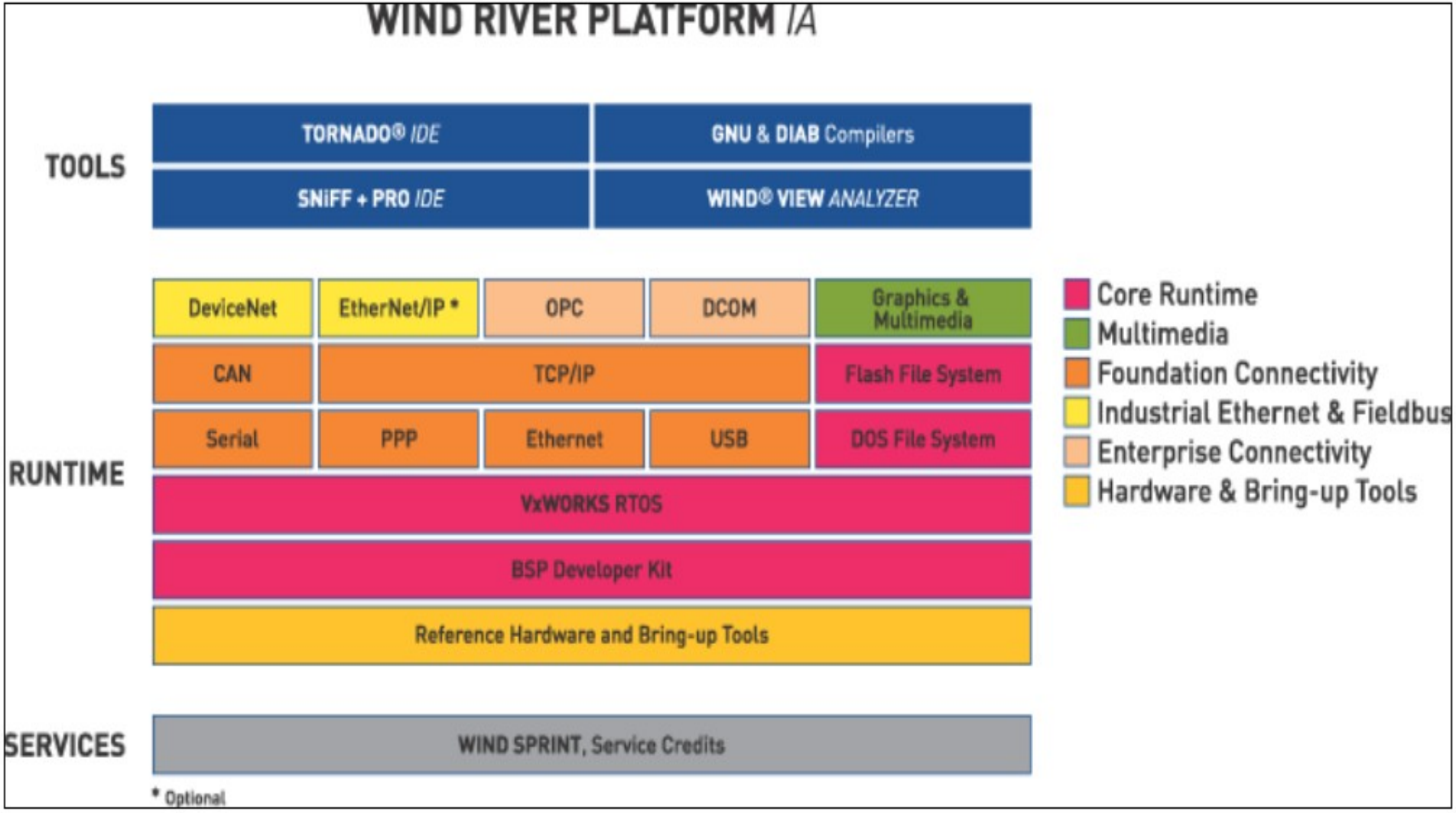
Embedded OS



Standard OS



Example: WindRiver Platform Industrial Automation



Embedded operating systems
- Characteristics: Protection is optional -

Protection mechanisms not always necessary:

ES typically designed for a single purpose,

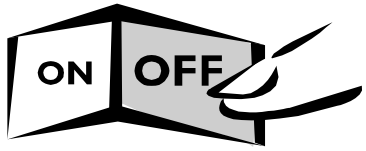
untested programs rarely loaded, SW considered reliable.

Privileged I/O instructions not necessary and tasks can do their own I/O.

Example: Let **`switch`** be the address of some switch
Simply use

```
load register,switch
```

instead of OS call.



However, protection mechanisms may be needed for safety and security reasons.

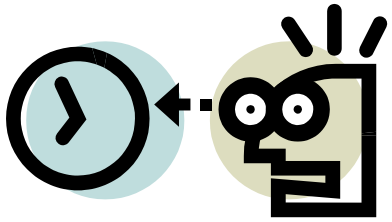
Embedded operating systems
- Characteristics: Interrupts not restricted to OS -

Interrupts can be employed by any process
For standard OS: serious source of unreliability.
Since

- embedded programs can be considered to be tested,
- since protection is not always necessary and
- since efficient control over a variety of devices is required,
- it is possible to let interrupts directly start or stop SW (by storing the start address in the interrupt table).
- More efficient than going through OS services.
- Reduced composability: if SW is connected to an interrupt, it may be difficult to add more SW which also needs to be started by an event.

Embedded operating systems
- Characteristics: Real-time capability -

Many embedded systems are real-time (RT) systems and, hence, the OSs used in these systems must be **real-time operating systems (RTOSs).**



RT operating systems
- Definition and Requirement 1: predictability -

Def.: *(A) real-time operating system is an operating system that supports the construction of real-time systems.*

The following are the three key requirements

1. **The timing behavior of the OS must be predictable.**
   ∀ services of the OS: Upper bound on the execution time!
   RTOSs must be timing-predictable:
   - short times during which interrupts are disabled,
   - (for hard disks:) contiguous files to avoid unpredictable head movements.

[Takada, 2001]

Real-time operating systems
- Requirement 2: Managing timing -

2. OS should manage the timing and scheduling

- OS possibly has to be aware of task deadlines; (unless scheduling is done off-line).
- Frequently, the OS should provide precise time services with high resolution.

[Takada, 2001]

Time

Time plays a central role in “real-time” systems

Physical time: real numbers

Computers: mostly discrete time

- Relative time: clock ticks in some resolution
- Absolute time: wall clock time
  - **International atomic time TAI**
    (french: *temps atomic internationale*)
    Free of any artifacts.
  - **Universal Time Coordinated (UTC)**
    UTC is defined by astronomical standards

TAI and UTC identical on Jan. 1st, 1958.

30 seconds had to be added since then.

Not without problems: New Year may start twice per night.

Internal synchronization

- Synchronization with one master clock
  - Typically used in startup-phases
- Distributed synchronization:
  1. Collect information from neighbors
  2. Compute correction value
  3. Set correction value

Precision of step 1 depends on how information is collected:

- Application level: ~500 µs to 5 ms
- Operation system kernel: 10 µs to 100 µs
- Communication hardware: < 10 µs

External synchronization

External synchronization guarantees consistency with actual physical time.

Trend is to use GPS for ext. synchronization

GPS offers TAI and UTC time information.

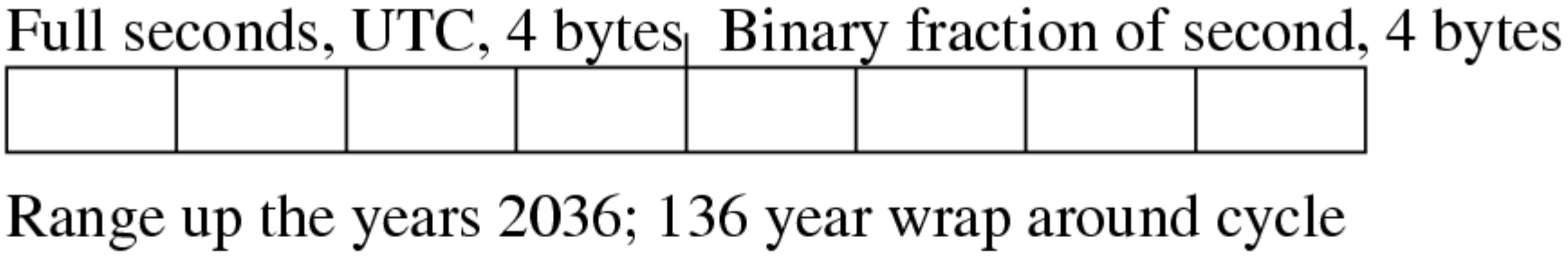
Resolution is about 100 ns.

GPS mouse

Problems with external synchronization

Problematic from the perspective of fault tolerance:
Erroneous values are copied to all stations.
Consequence: Accepting only small changes to local time.

Many time formats too restricted;
e.g.: NTP protocol includes only years up to 2036



For time services and global synchronization of clocks see Kopetz, 1997.

Real-time operating systems
- Requirement 3: Speed -

3. **The OS must be fast**

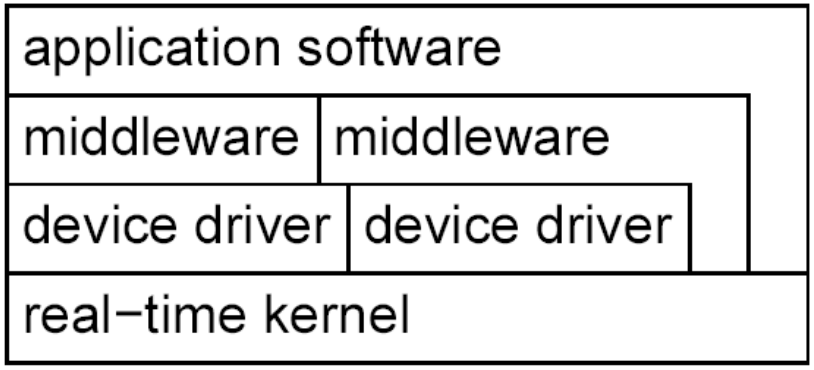
   Practically important.

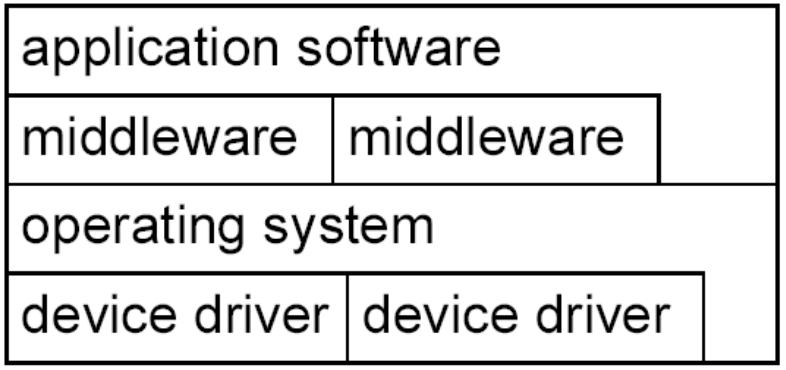
[Takada, 2001]

RTOS-Kernels

Distinction between

- real-time kernels and modified kernels of standard OSes.





Distinction between

- general RTOSs and RTOSs for specific domains,
- standard APIs (e.g. POSIX RT-Extension of Unix, ITRON, OSEK) or proprietary APIs.

Functionality of RTOS-Kernels

Includes

- processor management,
- memory management,
- and timer management;

} resource management

- task management (resume, wait etc),
- inter-task communication and synchronization.

Classes of RTOSes:
1. Fast proprietary kernels

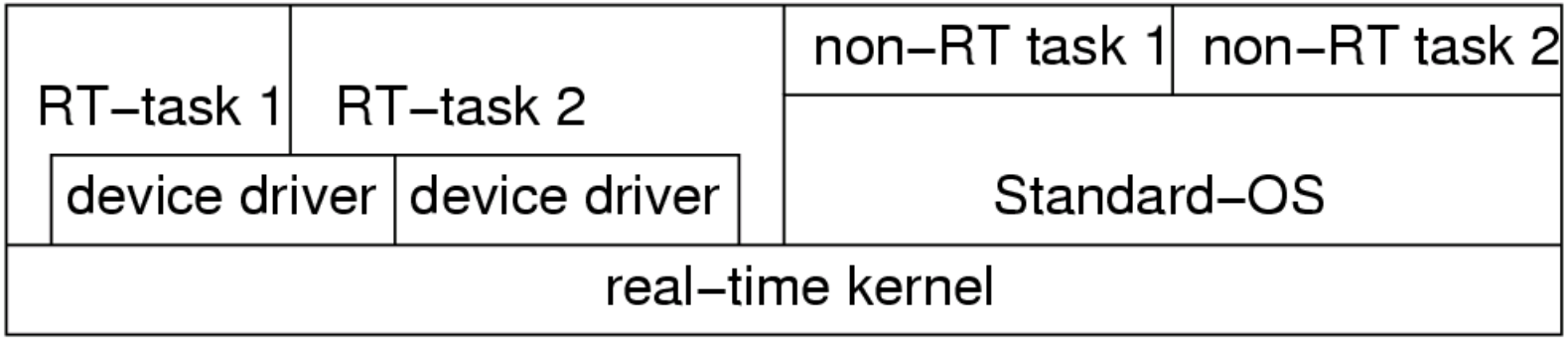
For complex systems, these kernels are inadequate, because they are designed to be fast, rather than to be predictable in every respect

[R. Gupta, UCI/UCSD]

Examples include: QNX, PDOS, VCOS, VTRX32, VxWORKS.

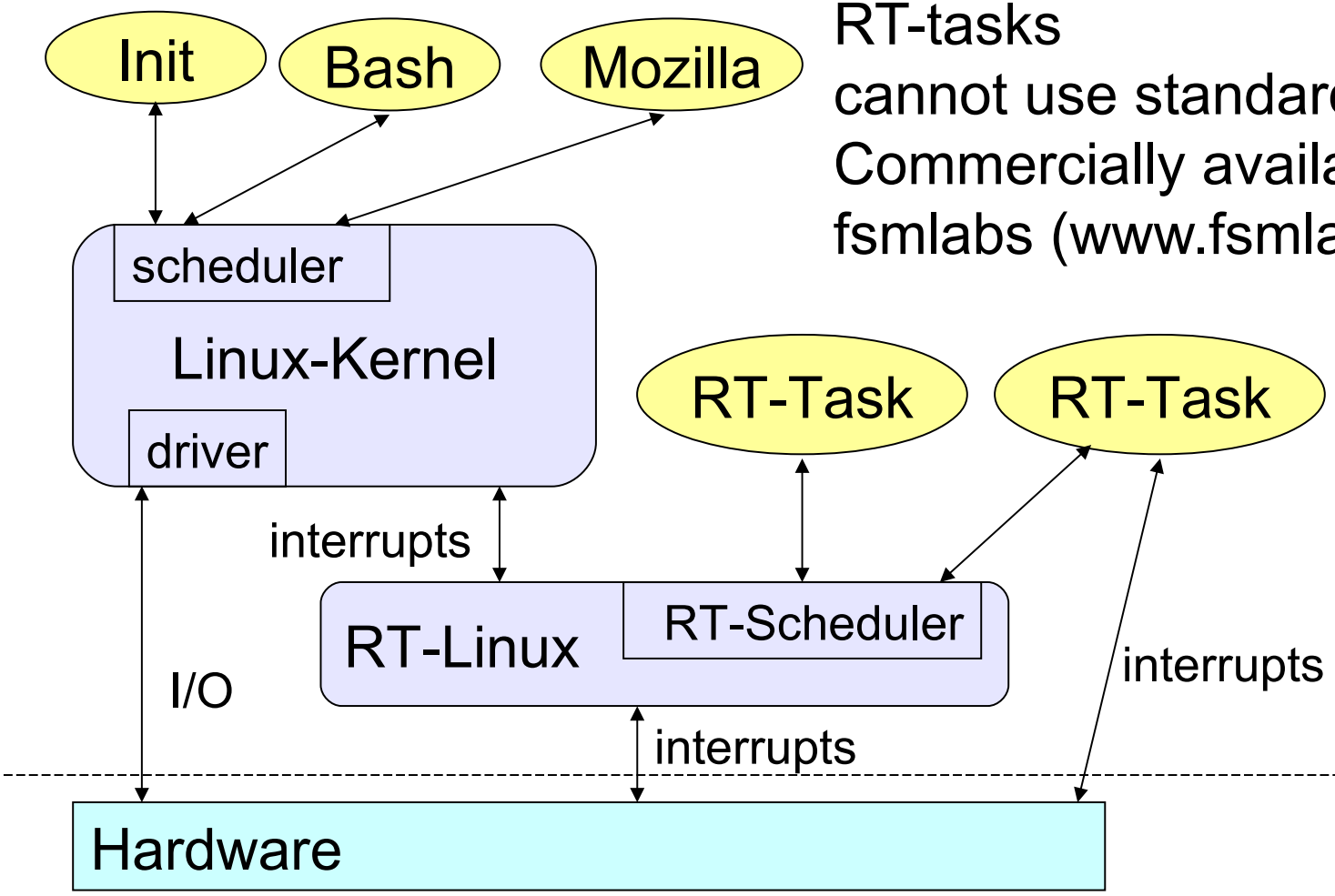
Classes of RTOSs:
2. RT extensions to standard OSs

Attempt to exploit comfortable main stream OS.
RT-kernel running all RT-tasks.
Standard-OS executed as one task.



+ Crash of standard-OS does not affect RT-tasks;
- RT-tasks cannot use Standard-OS services;
  less comfortable than expected

Example: RT-Linux



RT-tasks
cannot use standard OS calls.
Commercially available from
fsmlabs (www.fsmlabs.com)

Example (2): RTAI – Real Time Application Interface

https://www.rtai.org/

Fixes to many of the sources for unpredictability in Linux

Hardware abstraction layer in between hardware and Linux

Evaluation

According to Gupta, trying to use a version of a standard OS:

not the correct approach because too many basic and inappropriate underlying assumptions still exist such as optimizing for the average case (rather than the worst case), ... ignoring most if not all semantic information, and independent CPU scheduling and resource allocation.

Dependences between tasks not frequent for most applications of std. OSs & therefore frequently ignored.

Situation different for ES since dependences between tasks are quite common.

Classes of RTOSs:
3. Research trying to avoid limitations

Research systems trying to avoid limitations.
Include MARS, Spring, MARUTI, Arts, Hartos, DARK, and Melody

Research issues [Takada, 2001]:

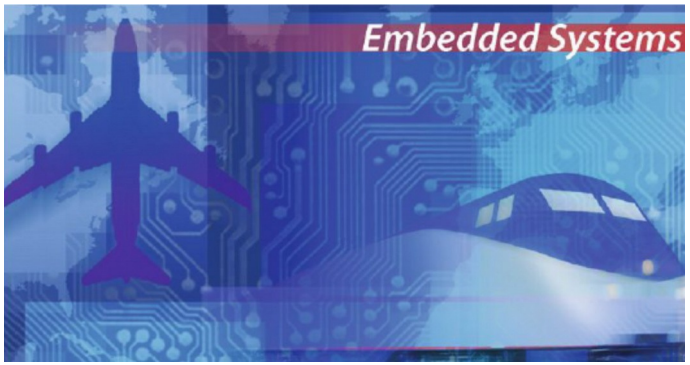
- low overhead memory protection,
- temporal protection of computing resources
- RTOSes for on-chip multiprocessors
- support for continuous media
- quality of service (QoS) control.

technische universität
dortmund

fakultät für informatik
informatik 12

Resource Access Protocols

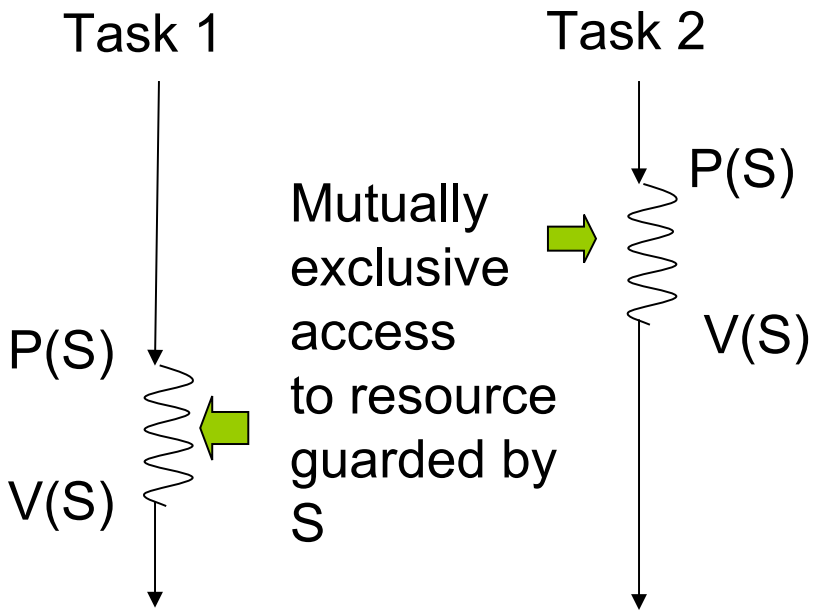
Peter Marwedel
TU Dortmund,
Informatik 12



2013 年 11 月 26 日

Resource access protocols

Critical sections: sections of code at which exclusive access to some resource must be guaranteed.
Can be guaranteed with semaphores S or “mutexes”.



P(S) checks semaphore to see if resource is available and if yes, sets S to “used“. Uninterruptible operations! If no, calling task has to wait.

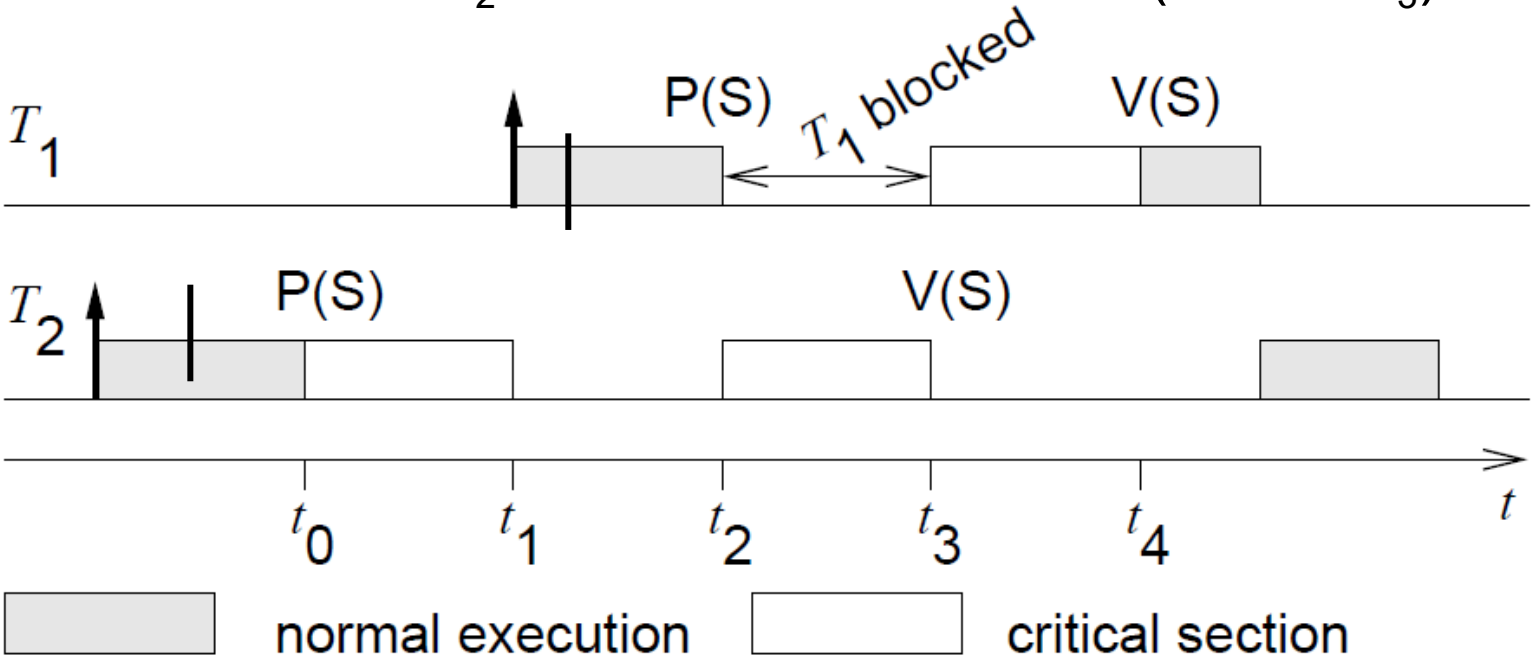
V(S): sets S to “unused“ and starts sleeping task (if any).

Blocking due to mutual exclusion

Priority T_1 assumed to be > than priority of T_2.

If T_2 requests exclusive access first (at t_0),

T_1 has to wait until T_2 releases the resource (at time t_3):



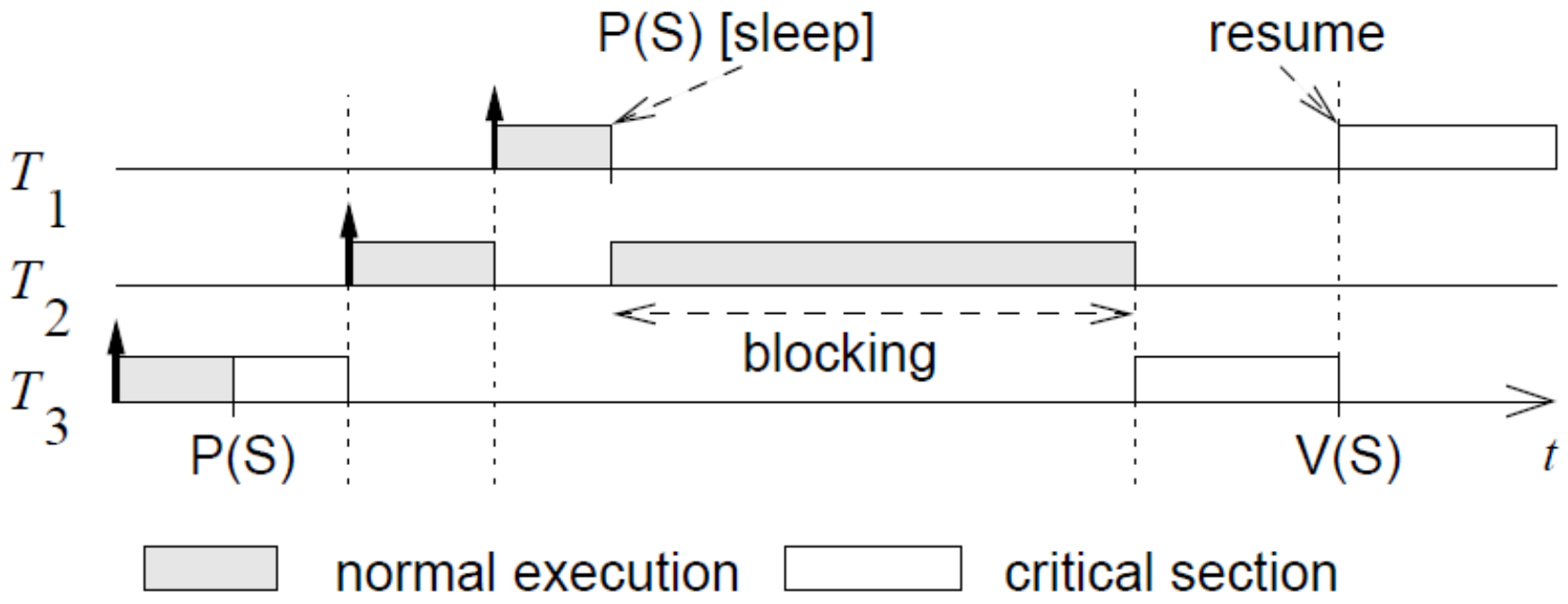
For 2 tasks:

blocking is bounded by the length of the critical section

Blocking with >2 tasks can exceed the length of any critical section

Priority of T_1 > priority of T_2 > priority of T_3.

T_2 preempts T_3: T_2 can prevent T_3 from releasing the resource.



Priority inversion!

The MARS Pathfinder problem (1)

"But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as "software glitches" and "the computer was trying to do too many things at once"." …

mars.jpl.nasa.gov

https://web.archive.org/web/20150612010836/http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/
was: http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html

The MARS Pathfinder problem (2)

“VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.”

“Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft.”

- A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).”

http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html

The MARS Pathfinder problem (3)

- The meteorological data gathering task ran as an infrequent, low priority thread, …
  When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. ..
- The spacecraft also contained a communications task that ran with medium priority.

☞ High priority: retrieval of data from shared memory

Medium priority: communications task

Low priority: thread collecting meteorological data

http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html

The MARS Pathfinder problem (4)

"… However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread.

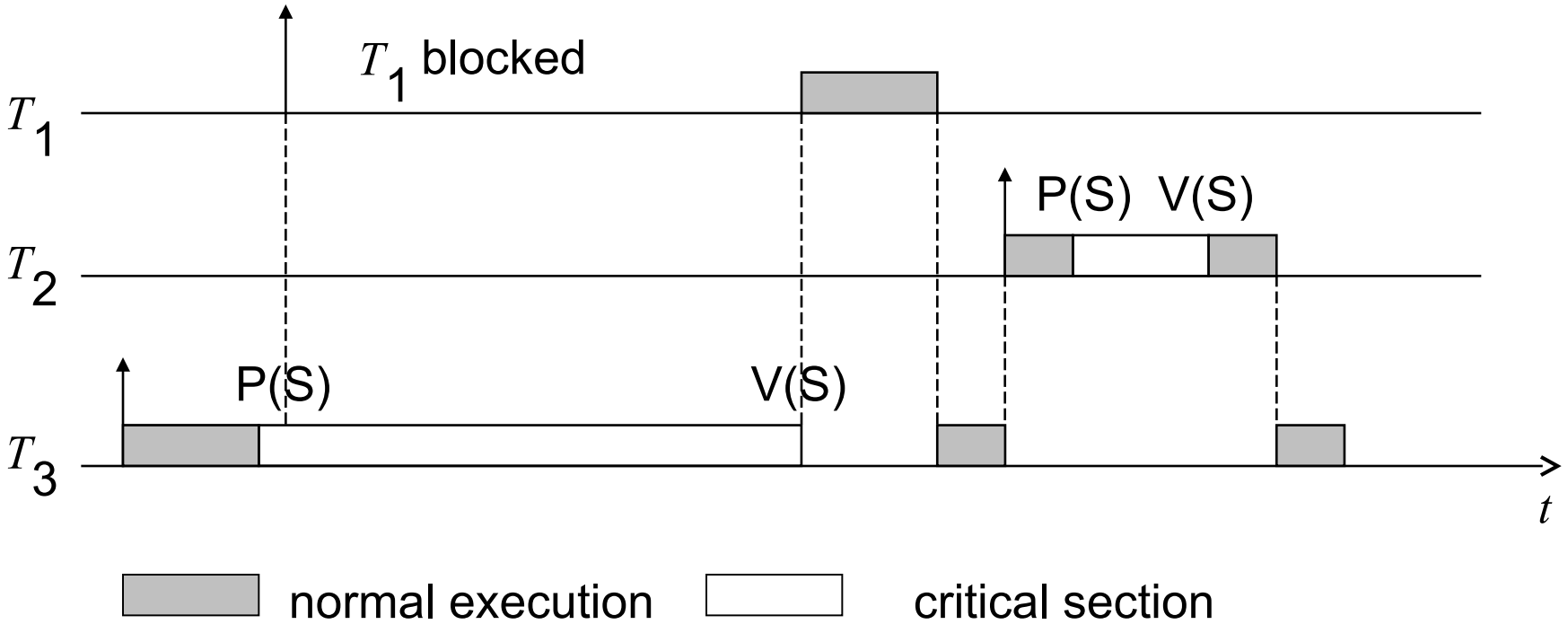
In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running.

After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset."

http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html

Solutions

Disallow preemption during the execution of all critical sections. Simple, but creates unnecessary blocking as unrelated tasks may be blocked.

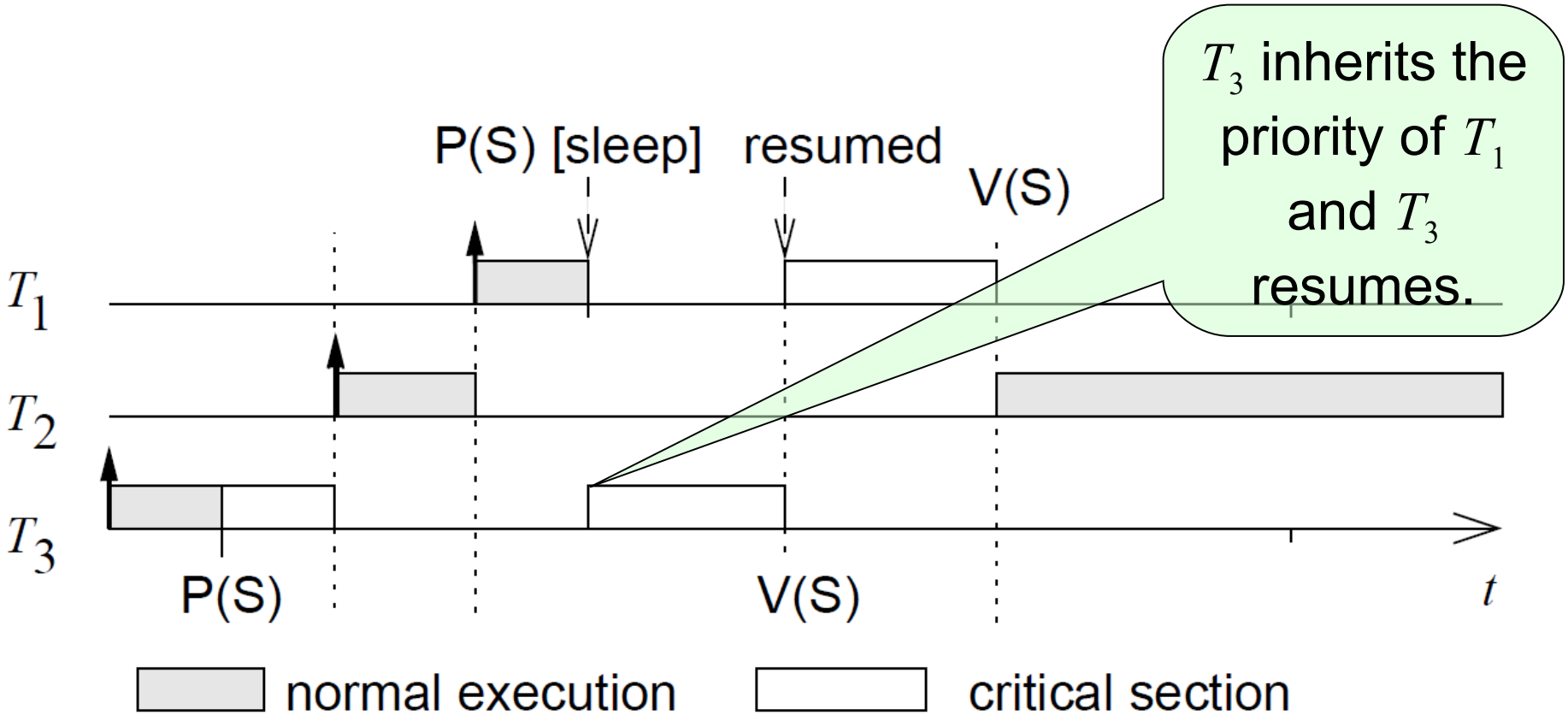


Coping with priority inversion: the priority inheritance protocol

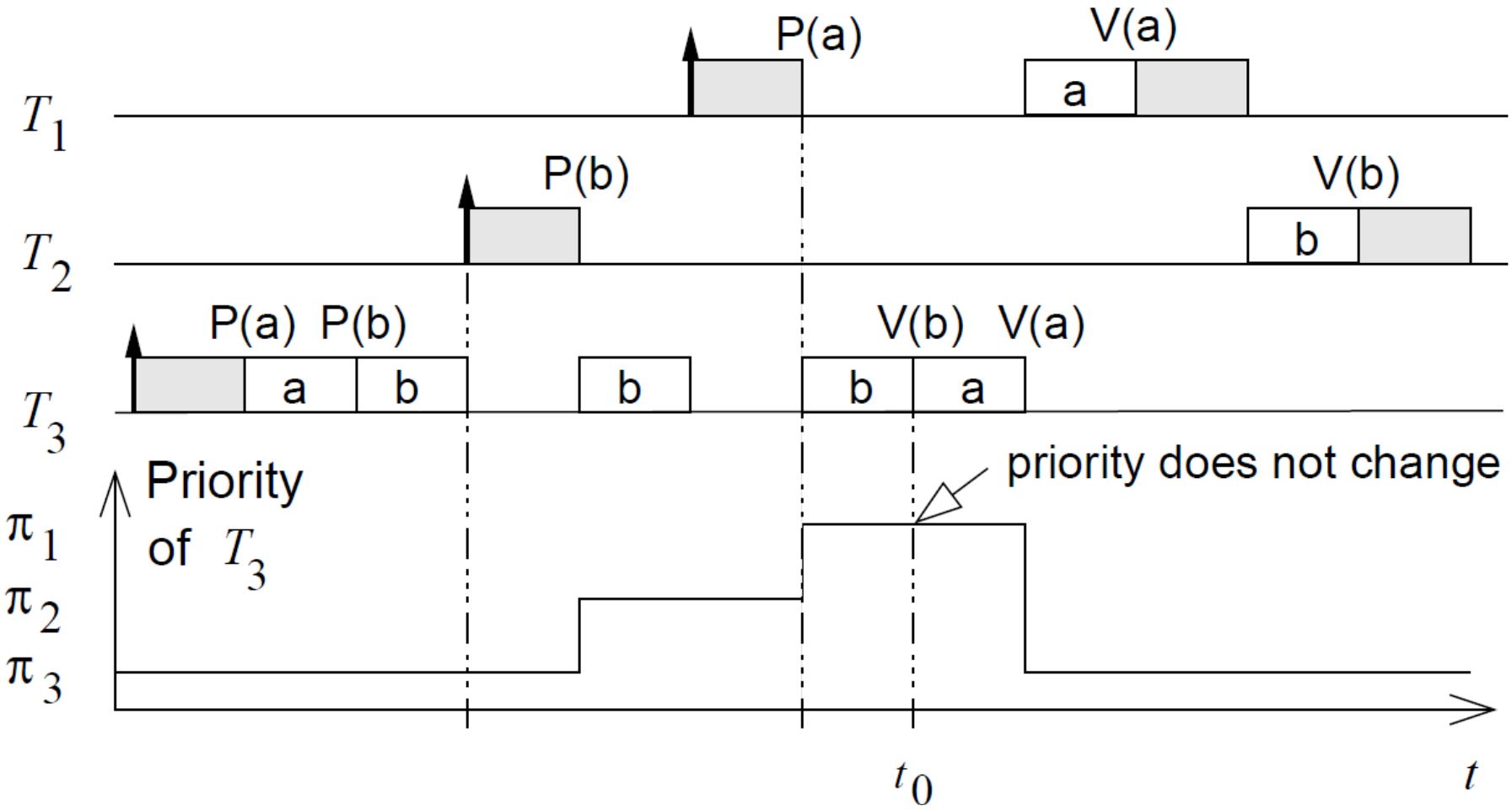
- Tasks are scheduled according to their active priorities.
  Tasks with the same priorities are scheduled FCFS.
- If task T_1 executes P(S) & exclusive access granted to T_2:
  T_1 will become blocked.
  If priority(T_2) < priority(T_1): T_2 inherits the priority of T_1.
  ☞ T_2 resumes.
  Rule: tasks inherit the highest priority of tasks blocked by it.
- When T_2 executes V(S), its priority is decreased to the highest priority of the tasks blocked by it.
  If no other task blocked by T_2: priority(T_2):= original value.
  Highest priority task so far blocked on S is resumed.
- Transitive: if T_2 blocks T_1 and T_1 blocks T_0,
  then T_2 inherits the priority of T_0.

Example

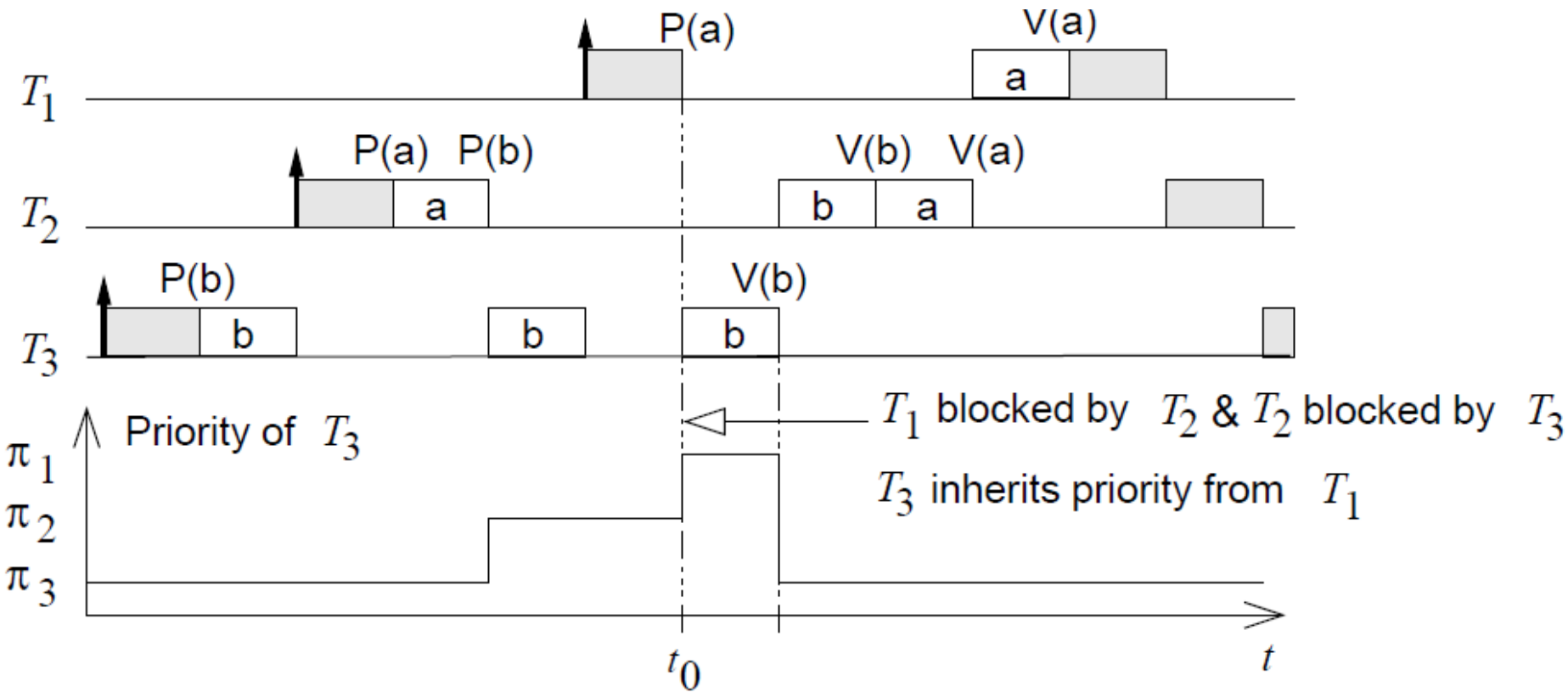
How would priority inheritance affect our example with 3 tasks?



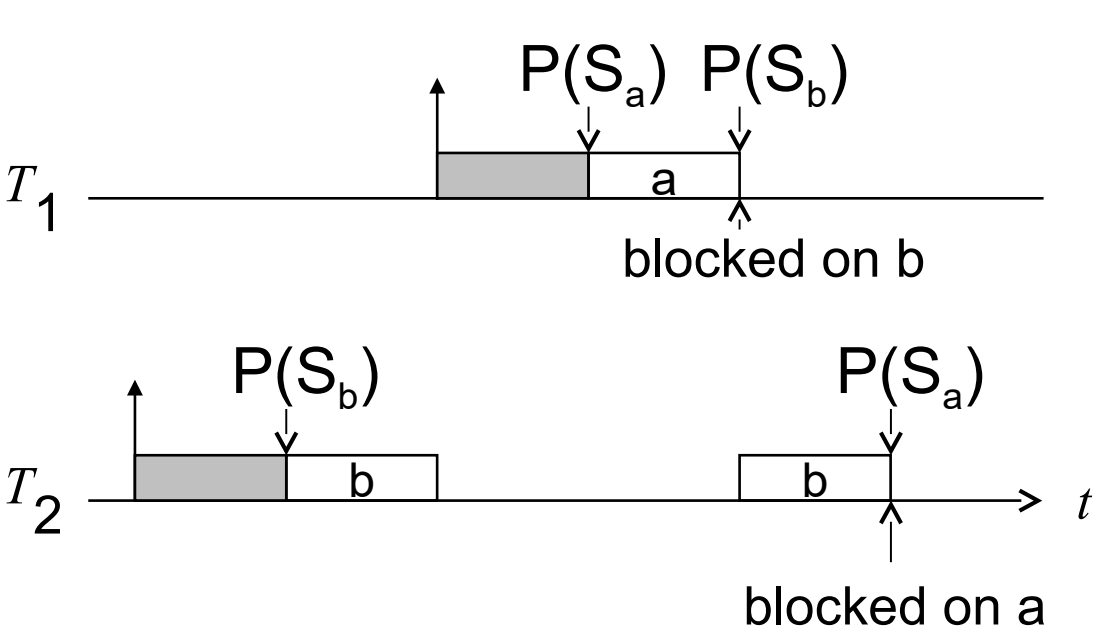
Nested critical sections

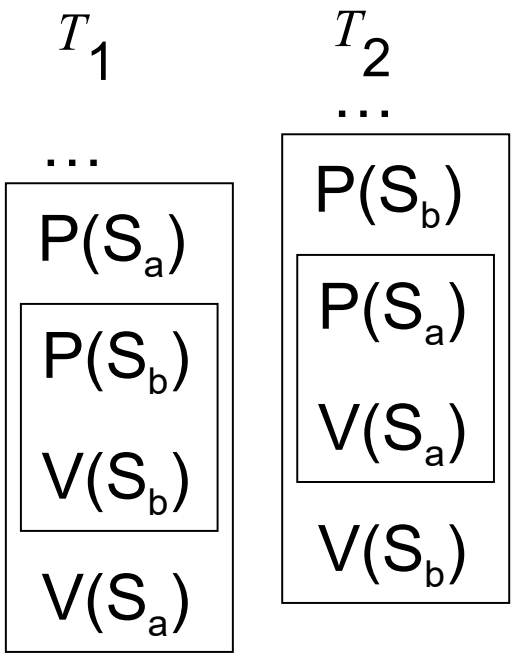


Transitiveness of priority inheritance



Deadlock is possible





normal execution critical section

Problem exists also when no priority inheritance is used

Priority inversion on Mars

Priority inheritance also solved the Mars Pathfinder problem: the VxWorks operating system used in the pathfinder implements a flag for the calls to mutex primitives. This flag allows priority inheritance to be set to “on”. When the software was shipped, it was set to “off”.

The problem on Mars was corrected by using the debugging facilities of VxWorks to change the flag to “on”, while the Pathfinder was already on the Mars [Jones, 1997].

mars.jpl.nasa.gov

Remarks on priority inheritance protocol

Possibly large number of tasks with high priority.

Possible deadlocks.

Ongoing debate about problems with the protocol:

> Victor Yodaiken: Against Priority Inheritance, Sept. 2004,
> http://www.fsmlabs.com/resources/white_papers/priority-inheritance/

Finds application in ADA: During *rendez-vous*,
task priority is set to the maximum.

Protocol for fixed set of tasks: priority ceiling protocol.

Summary

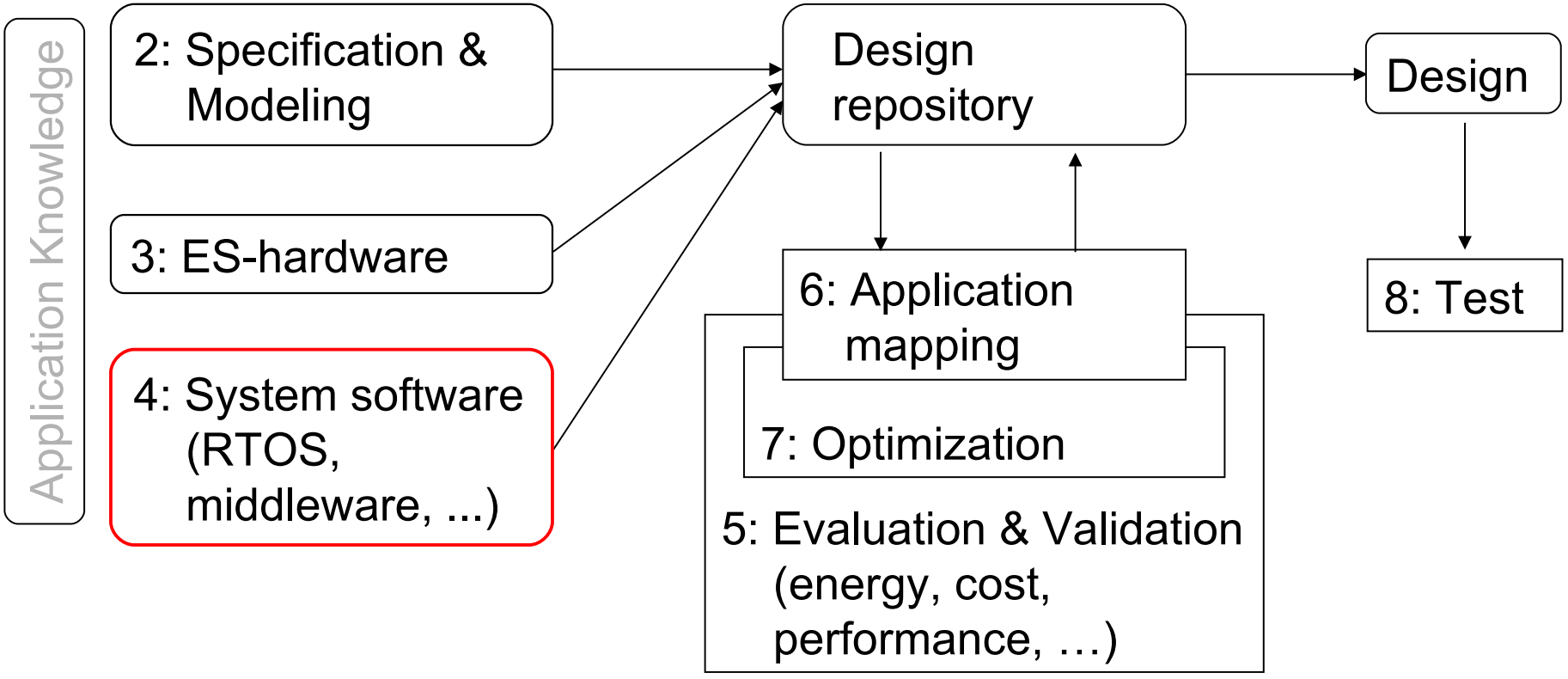
- General requirements for embedded operating systems
  - Configurability
  - I/O
  - Interrupts
- General properties of real-time operating systems
  - Predictability
  - Time services
  - Synchronization
  - Classes of RTOSs,
  - Device driver embedding
- Priority inversion
  - The problem
  - Priority inheritance

System Software (2)

Peter Marwedel
TU Dortmund,
Informatik 12

2013 年 11 月 26 日

Structure of this course



Numbers denote sequence of chapters

Increasing design complexity + Stringent time-to-market requirements ☞Reuse of components

Reuse requires knowledge from previous designs to be made available in the form of **intellectual property** (IP, for **SW & HW**).

- HW
- Operating systems
- Middleware (Communication, data bases, …)
- …

Priority Inheritance Protocol (PIP)
☞ Priority Ceiling Protocol (PCP)

The Priority Inheritance Protocol (PIP)

- does not prevent deadlocks
- can lead to chained blocking
  - Several lower priority tasks can block a higher priority task
- and has inherent static priorities of tasks

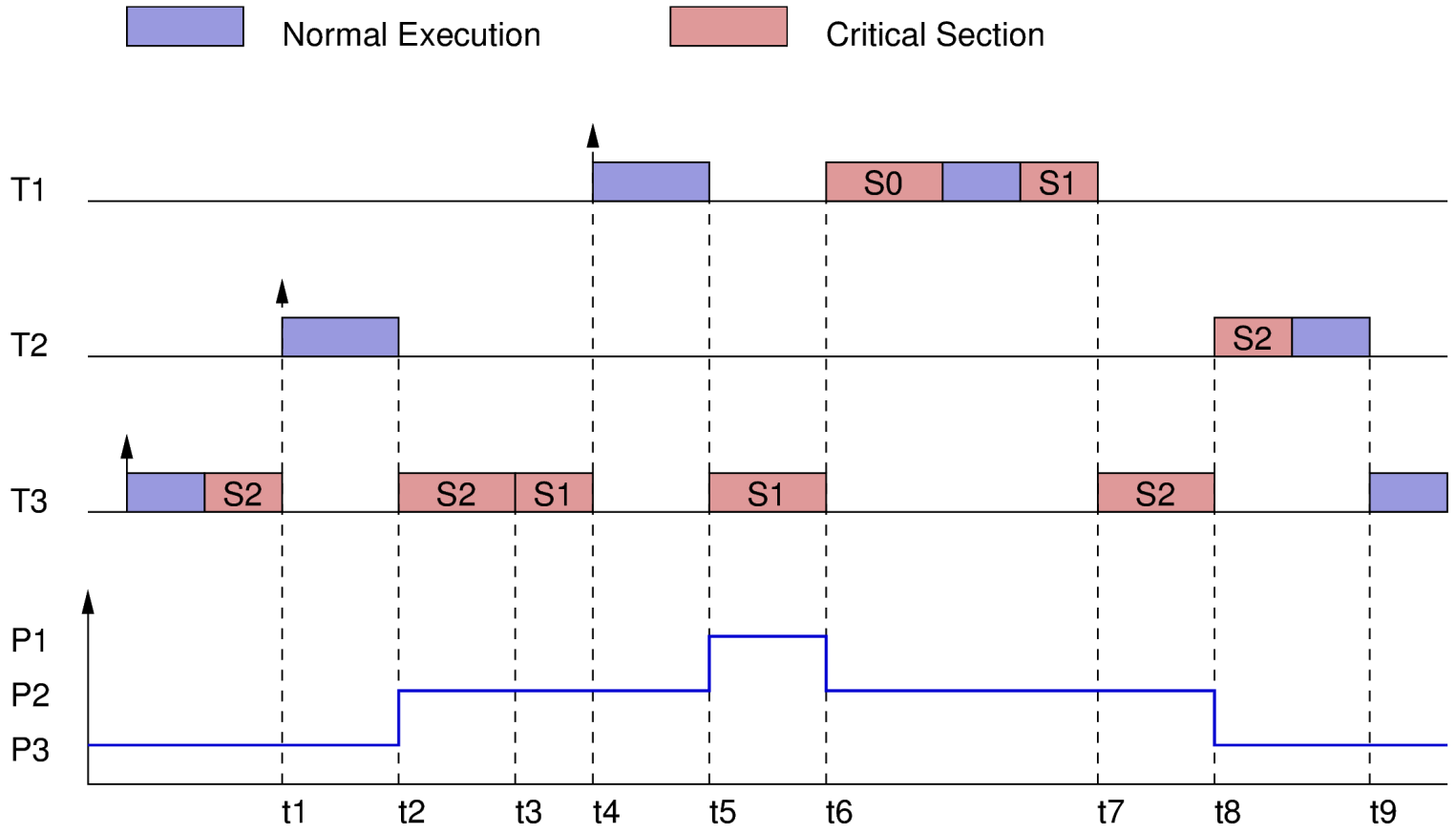
☞ The Priority Ceiling Protocol (PCP)

- avoids multiple blocking
- guarantees that, once a task has entered a critical section, it cannot be blocked by lower priority tasks until its completion.

PCP (1)

- A task is not allowed to enter a critical section if there are already locked semaphores which could block it eventually
- Hence, once a task enters a critical section, it can not be blocked by lower priority tasks until its completion.
- This is achieved by assigning priority ceiling.
- Each semaphore S_k is assigned a **priority ceiling** $C(S_k)$.
  It is the priority of the highest priority task that can lock S_k.
  This is a static value.

Priority Ceiling: Example



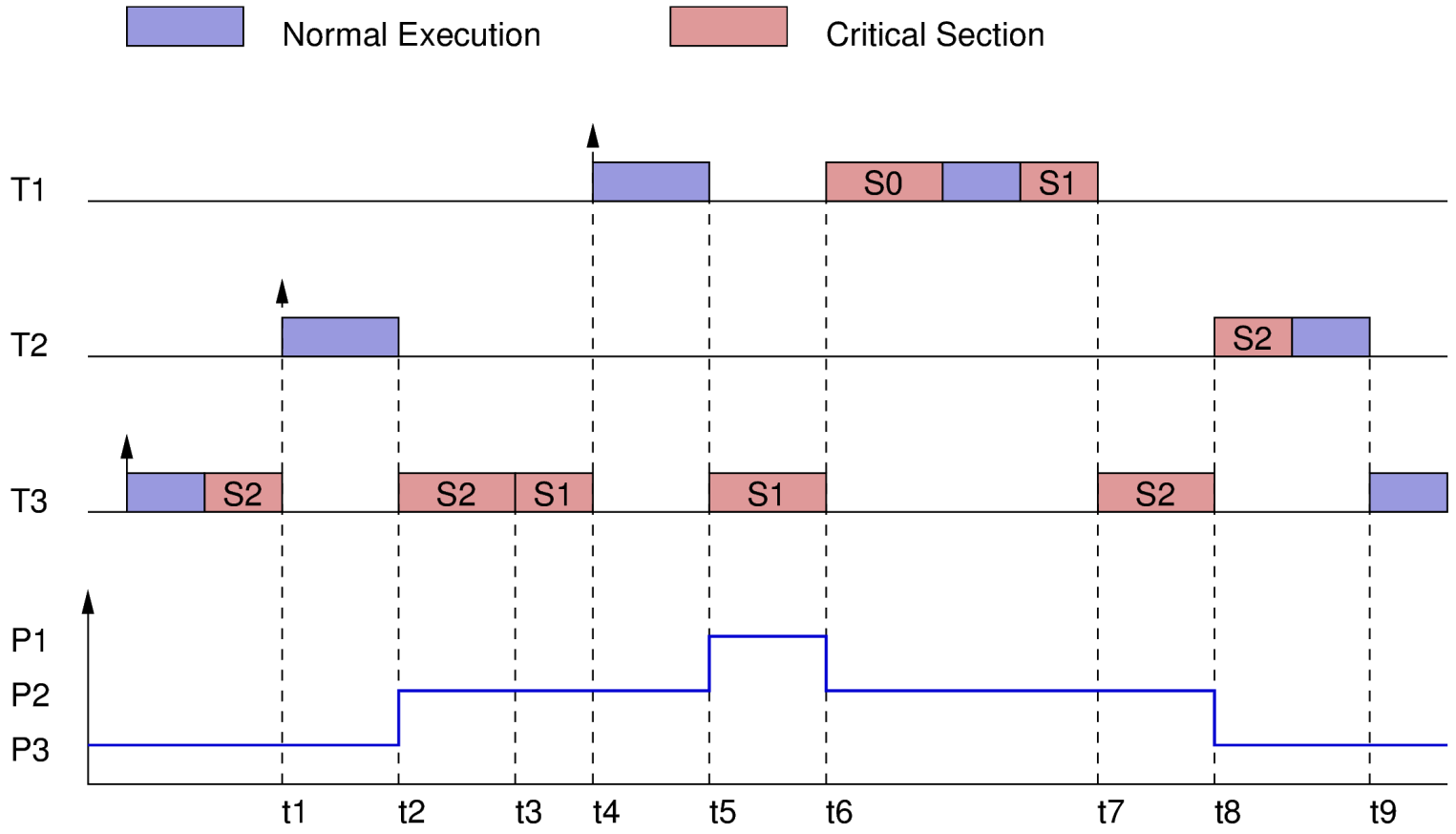
C(S0) = P1 C(S1) = P1 C(S2) = P2

Source (outdated): http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf

PCP (2)

- Suppose T is running and wants to lock semaphore S_k.
- T is allowed to lock S_k only if
  priority of T > priority ceiling $C(S^*)$ of the semaphore S^* where:
  - S^* is the semaphore with the highest priority ceiling among all the semaphores which are currently locked by jobs other than T.
  - In this case, T is said to blocked by the semaphore S^* (and the job currently holding S^*)
  - When T gets blocked by S^* then the priority of T is transmitted to the job T that currently holds S^*

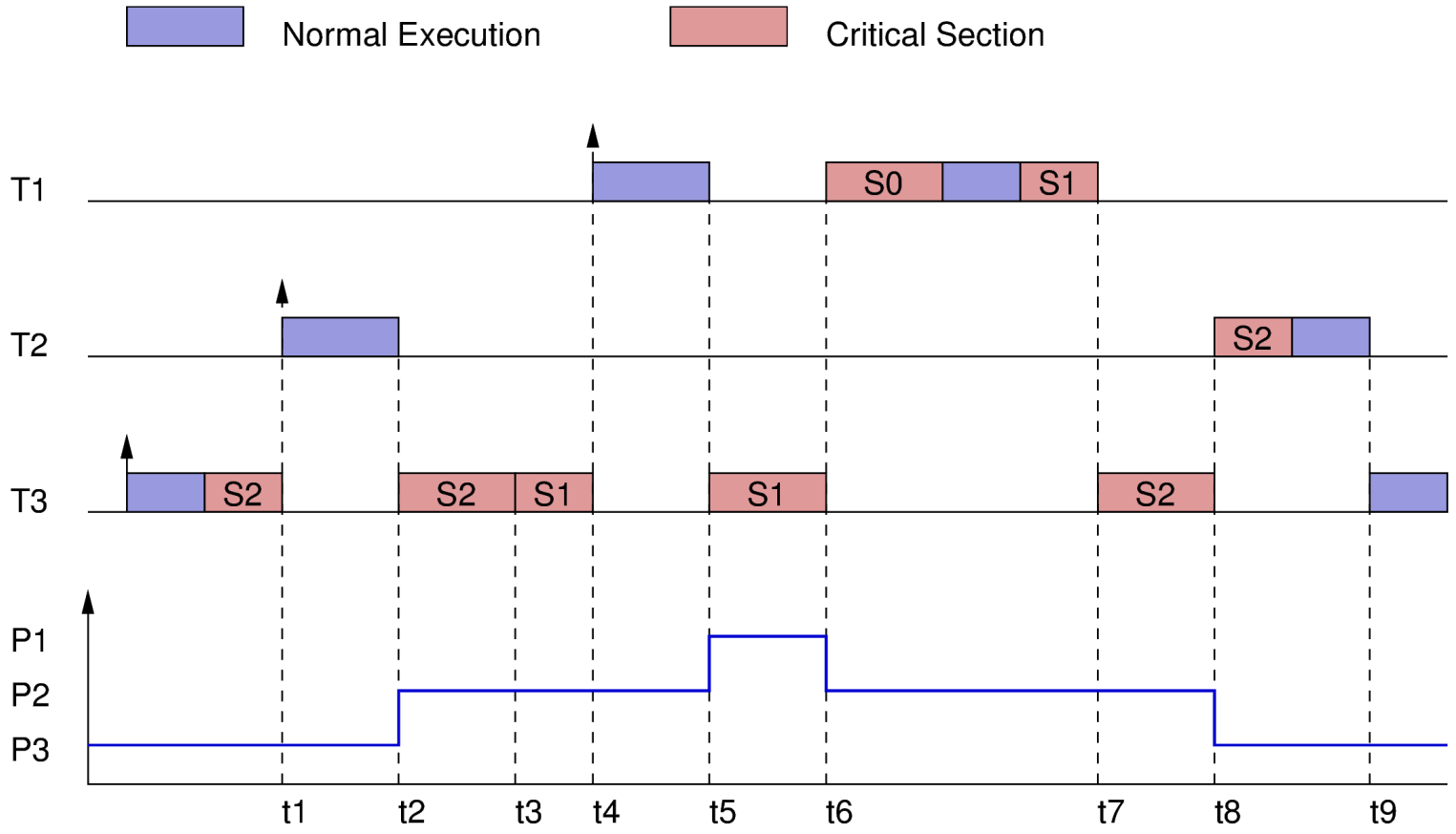
PCP: Example (1)



Source (outdated): http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf

t2: T2 can not lock S2. Currently T3 is holding S2 and C(S2) = P2 and the current priority of T2 is also P2

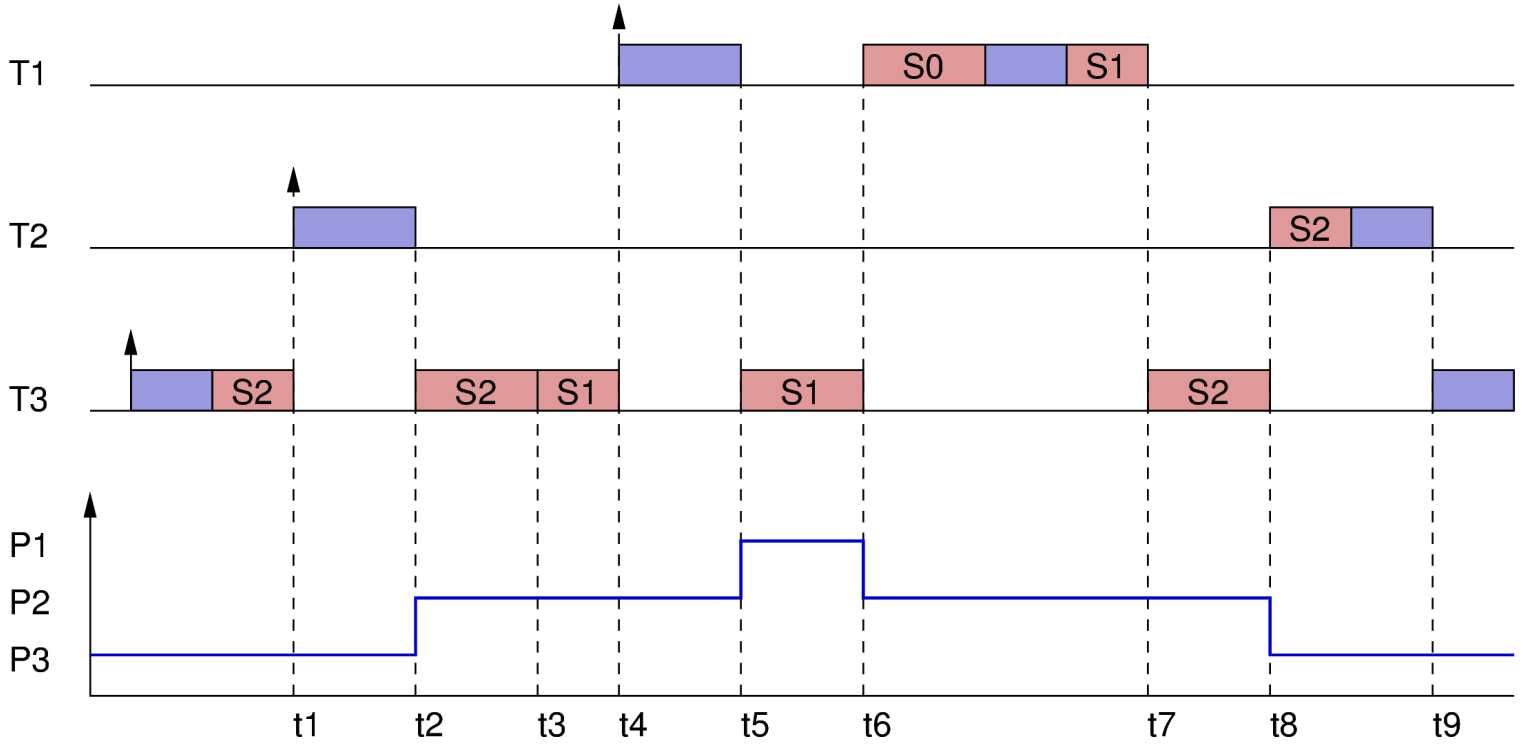
PCP: Example (2)



Source (outdated): http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf

t5: T1 can not lock S0. Currently T3 is holding S2 and S1 and C(S1) = P1 and the current priority of T1 is also P1.
The (inherited) priority of T3 is now P1

PCP: Example (3)



Source (outdated): http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf

t6: T3 unlocks S1. It awakens T1. But T3s inherited priority is now only P2 while P1 > C(S2) = P2. So T1 preempts T3 and run to completion.

t7: T3 resumes execution with priority P2

t8: T3 unlocks S2, goes back to its priority P3. T2 preempts T3, runs to completion

PCP (3)

- When T^* leaves a critical section guarded by S^* then it unlocks S^* and the highest priority job, if any, which is blocked by S^* is awakened
- The priority of T^* is set to the highest priority of the job that is blocked by some semaphore that T^* is still holding.
  If none, the priority of T^* is set to be its nominal one.

PCP: Example – no deadlocks (1)

Task name	T	Priority
A	50	10
B	500	9

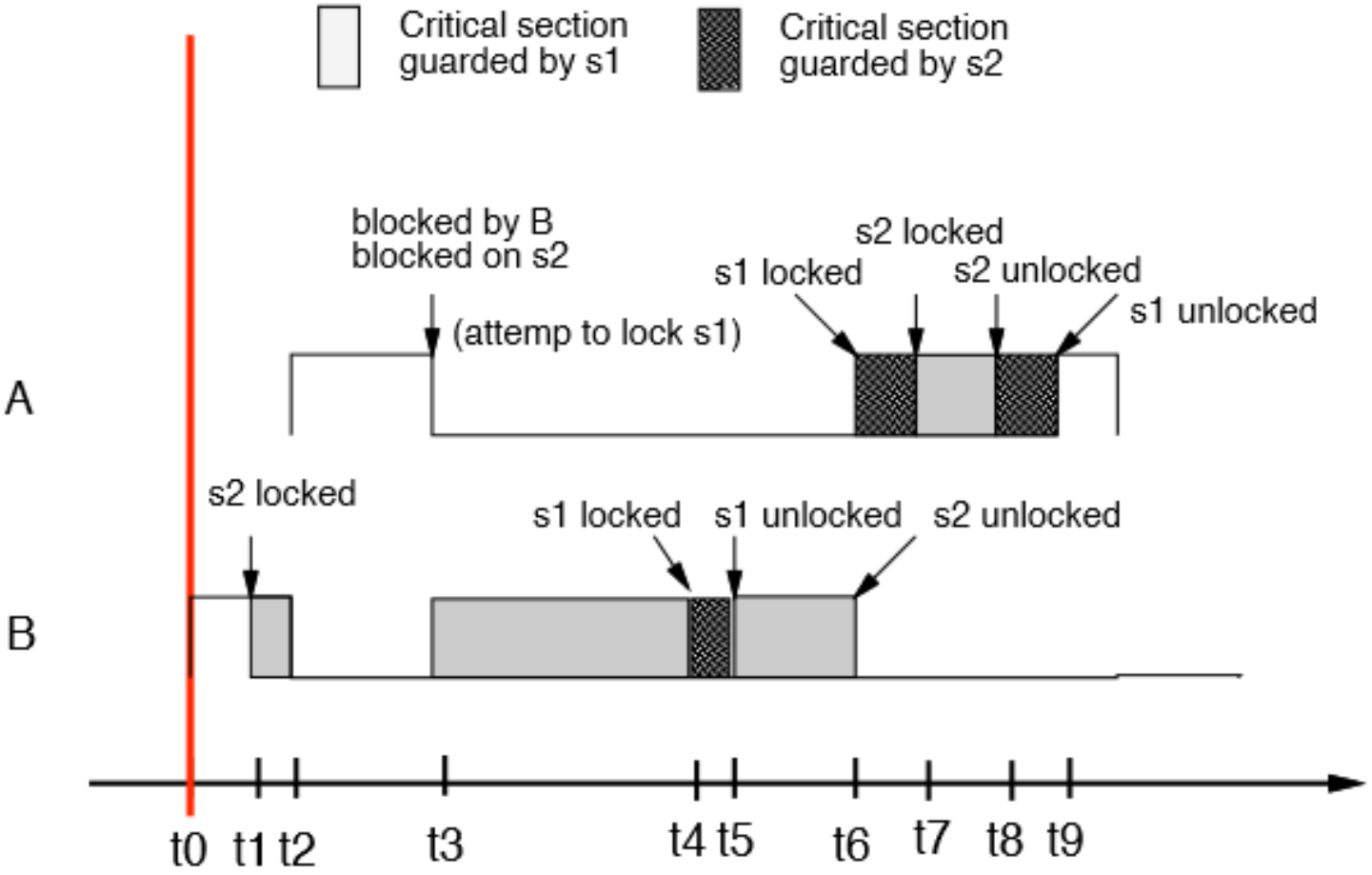
```
Task A        Task B

lock(s1)      lock(s2)
lock(s2)      lock(s1)
...           ...
unlock(s1)    unlock(s1)
unlock(s2)    unlock(s1)
```

$$ceil(s_1) = 10, ceil(s_2) = 10$$

Source: Lund University, course EDA 040, http://fileadmin.cs.lth.se/cs/Education/EDA040/lecture/RTP-F6b.pdf

PCP: Example – no deadlocks (2)



See http://fileadmin.cs.lth.se/cs/Education/EDA040/lecture/RTP-F6b.pdf for detailed explanation

PCP: Properties

- deadlock free (only changing priorities)
- a given task i is delayed at most once by a lower priority task
- the delay is a function of the time taken to execute the critical section
- Certain variants as to when the priority is changed

Extending PCP: Stack Resource Policy (SRP)

- SRP supports dynamic priority scheduling
- SRP blocks the task at the time it attempts to preempt.
- **Preemption level** l_i of task i: decreasing function of deadline (larger deadline ☞ easier to preempt) (Static)
- **Resource ceiling**: of a resource is the highest preemption level from among all tasks that may access that resource (Static)
- **System ceiling**: is the highest resource ceiling of all the resources which are currently blocked (dynamic, changes with resource accesses)

Stack Resource Policy (SRP)

A task can preempt another task if

- it has the highest priority
- and its preemption level is higher than the system ceiling

A task is not allowed to start until the resources currently available are sufficient to meet the maximum requirement of every task that could preempt it.

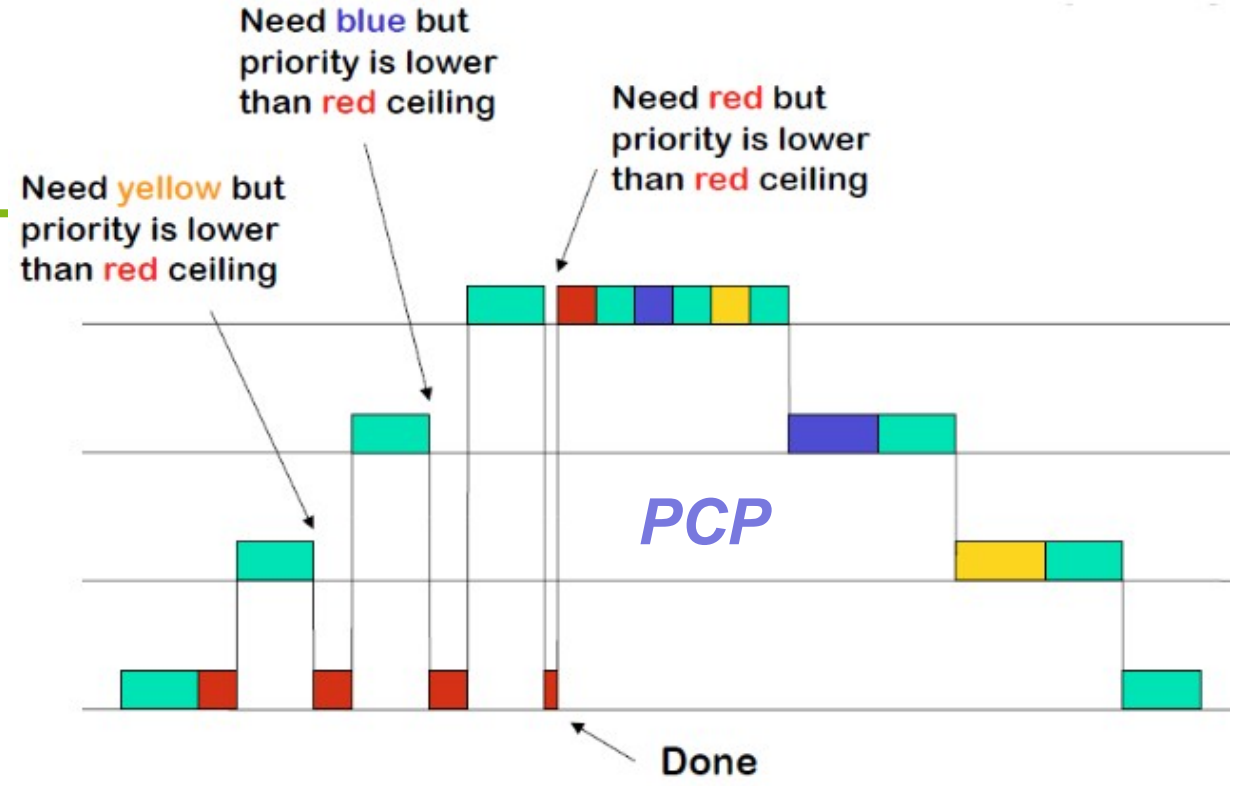
Why ***Stack*** Resource Policy? Tasks cannot be blocked by tasks with lower l_i, can resume only when the task completes.

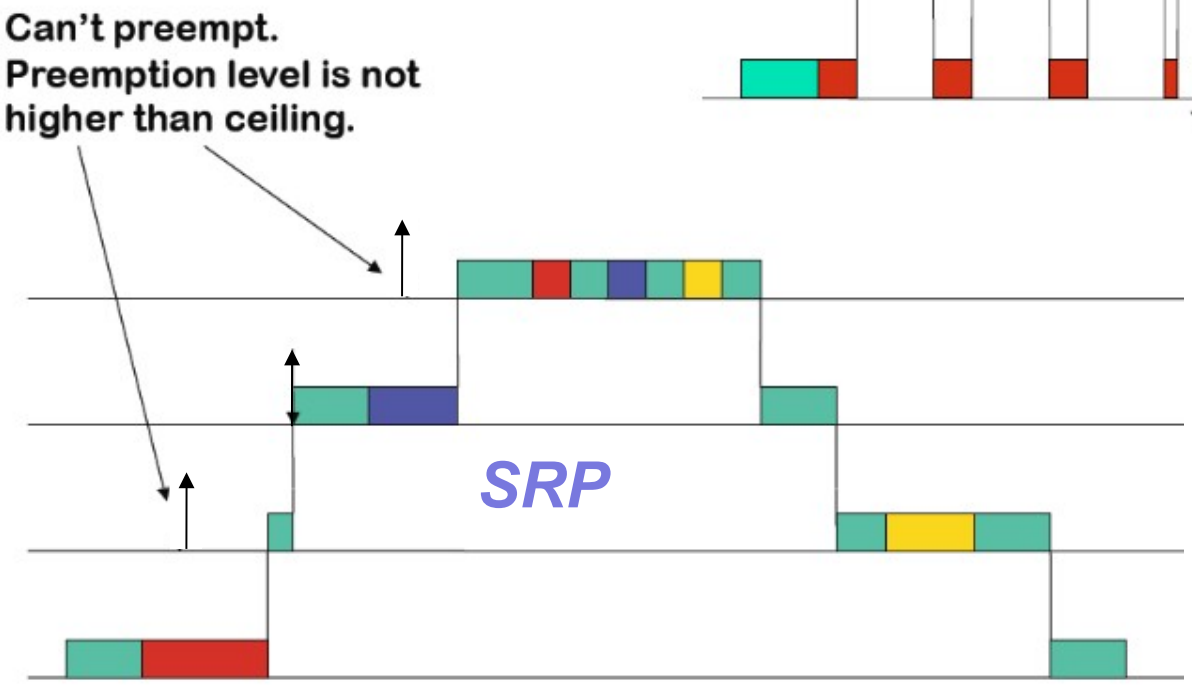
Tasks on the same l_i can share stack space.

More tasks on the same l_i ☞ higher stack space saving.

SRP vs. PCP

Less preemptions for SRP





Source (outdated): http://www.ida.liu.se/~unmbo/RTS_CUGS_files/Lecture3.pdf

Increasing design complexity + Stringent time-to-market requirements ☞Reuse of components

Reuse requires knowledge from previous designs to be made available in the form of **intellectual property** (IP, for **SW & HW**).

- HW
- Operating systems
- Middleware (Communication, data bases, …)
- …

Models of computation considered in this course

Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorith. ☞ Computer arch.		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC* …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java [libraries]	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

Pthreads (1)

- **Shared memory model**
- Consists of standard API
  - Originally used for single processor
  - Locks (mutex, read-write locks)

Based on W. Verachtert (IMEC): *Introduction to Parallelism,* tutorial, DATE 2008

Pthreads: Example

```
threads = (pthread_t *) malloc(n*sizeof(pthread_t));
pthread_attr_init(&pthread_custom_attr);
for (i=0;i<n; i++)
  pthread_create(&threads[i],
  &pthread_custom_attr, task, …);
for (i=0;i<n; i++) {
  pthread_mutex_lock(&mutex);
  <receive message>
  pthread_mutex_unlock(&mutex);
}
for (i=0;i<n; i++)
  pthread_join(threads[i], NULL);
```

```
void* task(void *arg) {
  …
  pthread_mutex_lock(&mutex);
  <send message>
  pthread_mutex_unlock(&mutex);
  return NULL
}
```

Based on W. Verachtert (IMEC): *Introduction to Parallelism,* tutorial, DATE 2008

Pthreads (2)

- Consists of standard API
  - Locks (mutex, read-write locks)
  - Condition variables
  - Completely explicit synchronization
  - Synchronization is very hard to program correctly
- Typically supported by a mixture of hardware (shared memory) and software (thread management)
- Exact semantics depends on the memory consistency model
- Support for efficient producer/consumer parallelism relies on murky parts of the model
- Pthreads can be used as back-end for other programming models (e.g. OpenMP)

Based on W. Verachtert (IMEC): *Introduction to Parallelism,* tutorial, DATE 2008

OpenMP

Implementations target shared memory hardware

Parallelism expressed using pragmas

- Parallel loops
  (`#pragma omp for {…}` ; focus: data parallelism)
- Parallel sections
- Reductions

Explicit

- Expression of parallelism (mostly explicit)

Implicit

- Computation partitioning
- Communication
- Synchronization
- Data distribution

Lack of control over partitioning can cause problems

Models of computation considered in this course

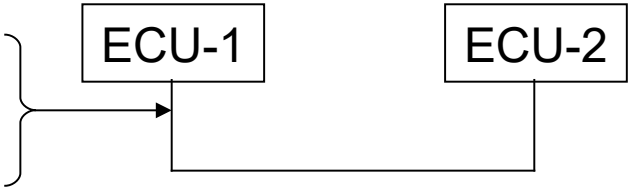
Communication/ local computations	Shared memory	Message passing Synchronous | Asynchronous	
Undefined components	Plain text, use cases | (Message) sequence charts		
Communicating finite state machines	StateCharts		SDL
Data flow	Scoreboarding + Tomasulo Algorith. ☞ Computer arch.		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC* …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java [libraries]	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation with centralized data structures

OSEK/VDX COM

OSEK/VDX COM

- is a special communication standard for the OSEK automotive OS Standard
- provides an “Interaction Layer” as an API for internal and external communication via a “Network Layer” and a “Data Link” layer (some requirements for these are specified)
- specifies the functionality, it is not an implementation.

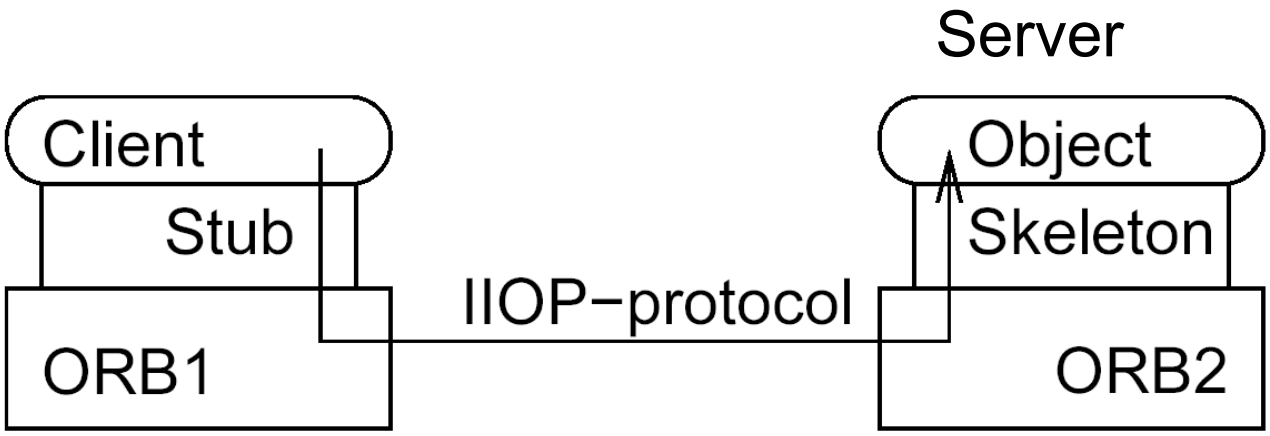


CORBA
(Common Object Request Broker Architecture)

Software package for access to remote objects;

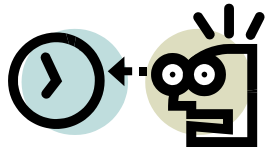
Information sent to Object Request Broker (ORB) via local stub.

ORB determines location to be accessed and sends information via the IIOP I/O protocol.



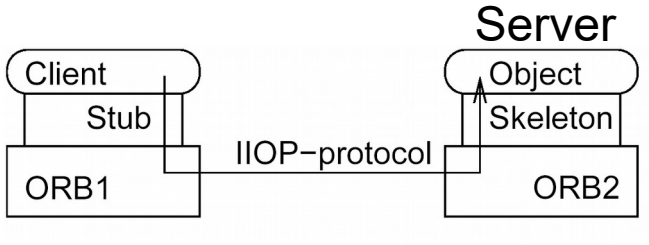
Access times unpredictable.

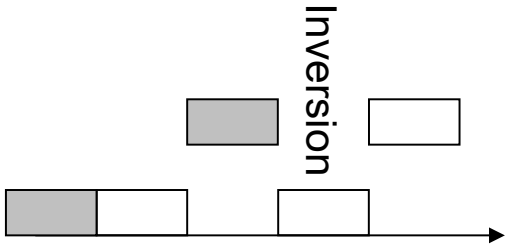
Real-time (RT-) CORBA



RT-CORBA

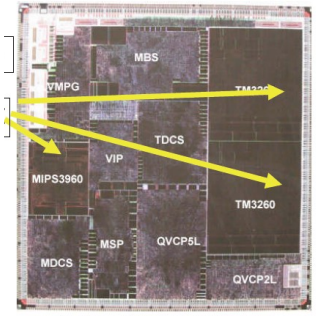
- provides *end-to-end predictability of timeliness in a fixed priority system*.
- *respects thread priorities between client and server for resolving resource contention*,
- provides thread priority management,
- provides priority inheritance,
- bounds latencies of operation invocations,
- provides pools of preexisting threads.





Message passing interface (MPI)

- Asynchronous/synchronous message passing
- Designed for high-performance computing
- Comprehensive, popular library
- Available on a variety of platforms
- Mostly for homogeneous multiprocessing
- Considered for MPSoC programs for ES;
- Includes many copy operations to memory (memory speed ~ communication speed for MPSoCs); Appropriate MPSoC programming tools missing.



MPI (1)

Sample blocking library call (for C)**:**

- MPI_Send(*buffer,count,type,dest,tag,comm*) where
  - *buffer*: address of data to be sent
  - *count*: number of data elements to be sent
  - *type*: data type of data to be sent (e.g. MPI_CHAR, MPI_SHORT, MPI_INT, …)
  - *dest*: process id of target process
  - *tag*: message id (for sorting incoming messages)
  - *comm*: communication context = set of processes for which destination field is valid
  - function result indicates success

MPI (2)

Sample non-blocking library call (for C)**:**

- MPI_Isend(*buffer,count,type,dest,tag,comm,request*) where
  - *buffer … comm:* same as above
  - *request*: unique "request number". "handle" can be used (in a WAIT type routine) to determine completion

Evaluation

Explicit

- Computation partitioning
- Communication
- Data distribution

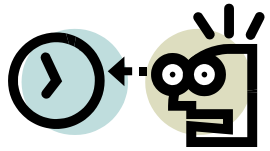
Implicit

- Synchronization (implied by communic., explicit possible)
- Expression of parallelism (implied)
- Communication mapping

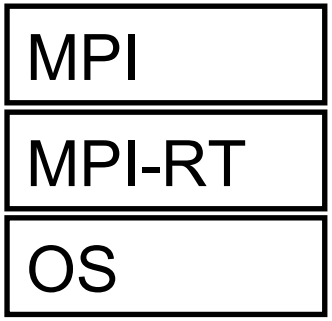
Properties

- Most things are explicit
- Lots of work for the user ("*assembly lang. for parallel prog*.")
- doesn't scale well when # of processors is changed heavily

RT-issues for MPI



- MPI/RT: a real-time version of MPI [MPI/ RT forum, 2001].
- MPI-RT does not cover issues such as thread creation and termination.
- MPI/RT is conceived as a potential layer between the operating system and standard (non real-time) MPI.



Summary

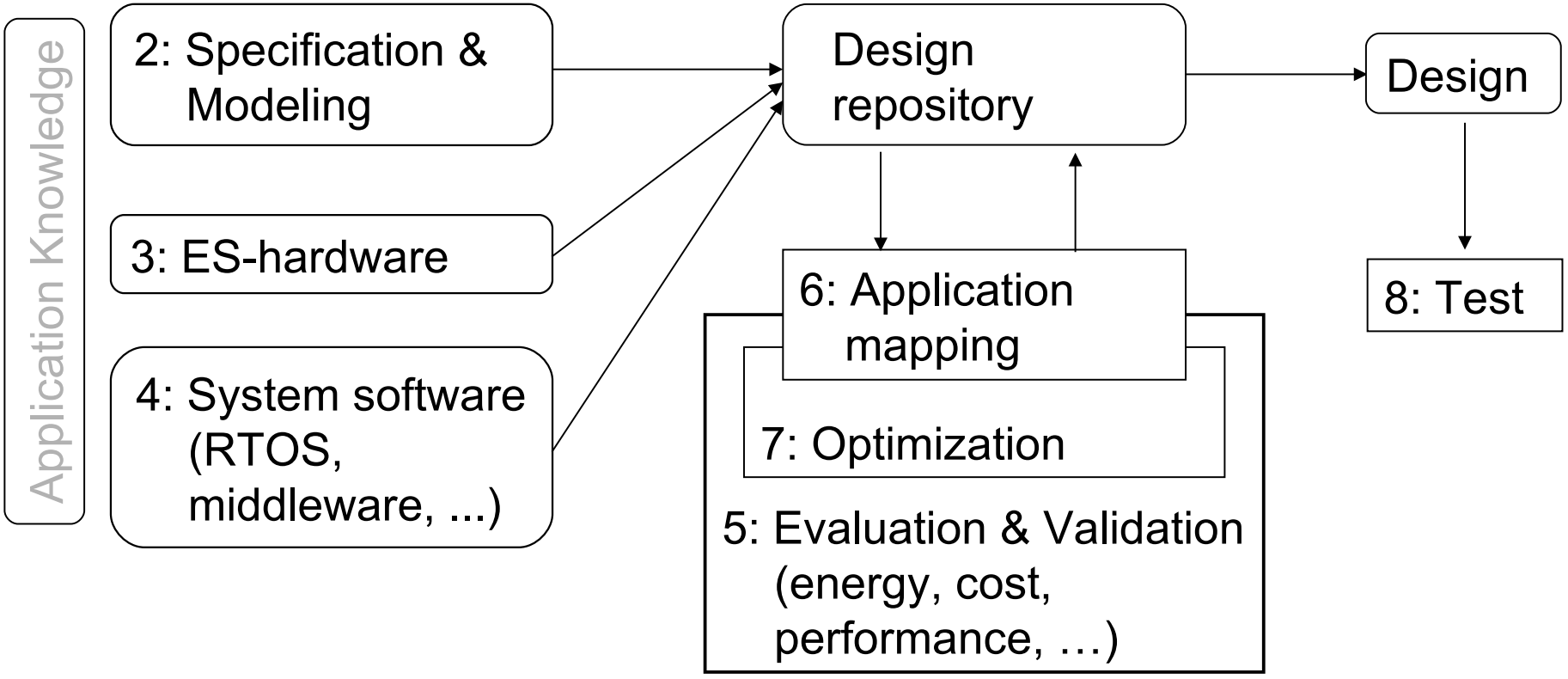
- Communication middleware
  - shared memory
    - Pthreads
    - OpenMP
  - message passing
    - OSEK/VDX COM
    - CORBA
    - MPI
    - JXTA
    - DPWS
- RT-Data bases (brief)

Evaluation and Validation

Peter Marwedel
TU Dortmund,
Informatik 12

2013 年 12 月 02 日

Structure of this course



Numbers denote sequence of chapters

Validation and Evaluation

Definition: *Validation* is the process of checking whether or not a certain (possibly partial) design is appropriate for its purpose, meets all constraints and will perform as expected (yes/no decision).

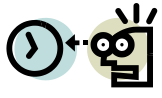
Definition: Validation with mathematical rigor is called *(formal) verification*.

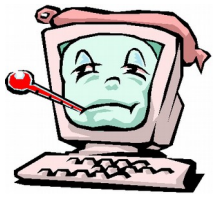
Definition: *Evaluation* is the process of computing quantitative information of some key characteristics of a certain (possibly partial) design.

How to evaluate designs according to multiple criteria?

Many different criteria are relevant for evaluating designs:

- Average & worst case delay
- power/energy consumption
- thermal behavior
- reliability, safety, security
- cost, size
- weight
- EMC characteristics
- radiation hardness, environmental friendliness, ..





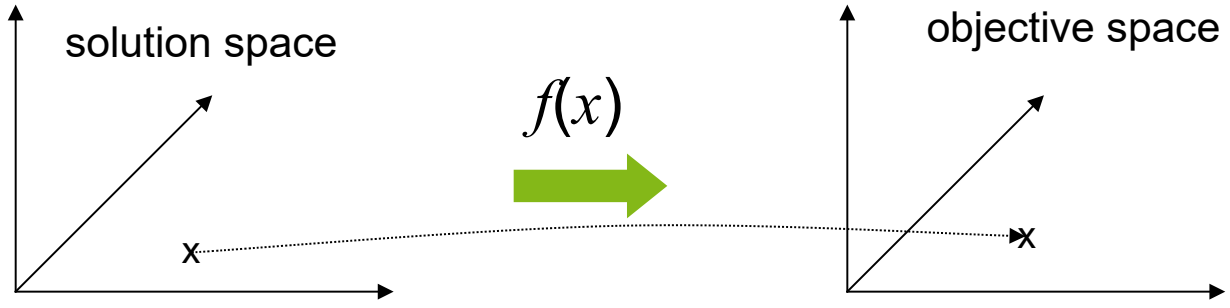




How to compare different designs?
(Some designs are “better” than others)

Definitions

- Let X: m-dimensional **solution space** for the design problem. Example: dimensions correspond to # of processors, size of memories, type and width of busses etc.
- Let F: n-dimensional **objective space** for the design problem. Example: dimensions correspond to average and worst case delay, power/energy consumption, size, weight, reliability, …
- Let $f(x)=(f_1(x),\ldots,f_n(x))$ where $x \in X$ be an **objective function.** We assume that we are using $f(x)$ for evaluating designs.



Pareto points (1)

- We assume that, for each objective, an order < and the corresponding order $\leq$ are defined.
- **Definition:**
  Vector $u=(u_1,...,u_n) \in F$ **dominates** vector $v=(v_1,...,v_n) \in F$
  $\Leftrightarrow$ is "better" than v with respect to one objective and not worse than v with respect to all other objectives:

$$\forall i \in \{1,\ldots,n\}: u_i \leq v_i \wedge$$
$$\exists i \in \{1,\ldots,n\}: u_i < v_i$$

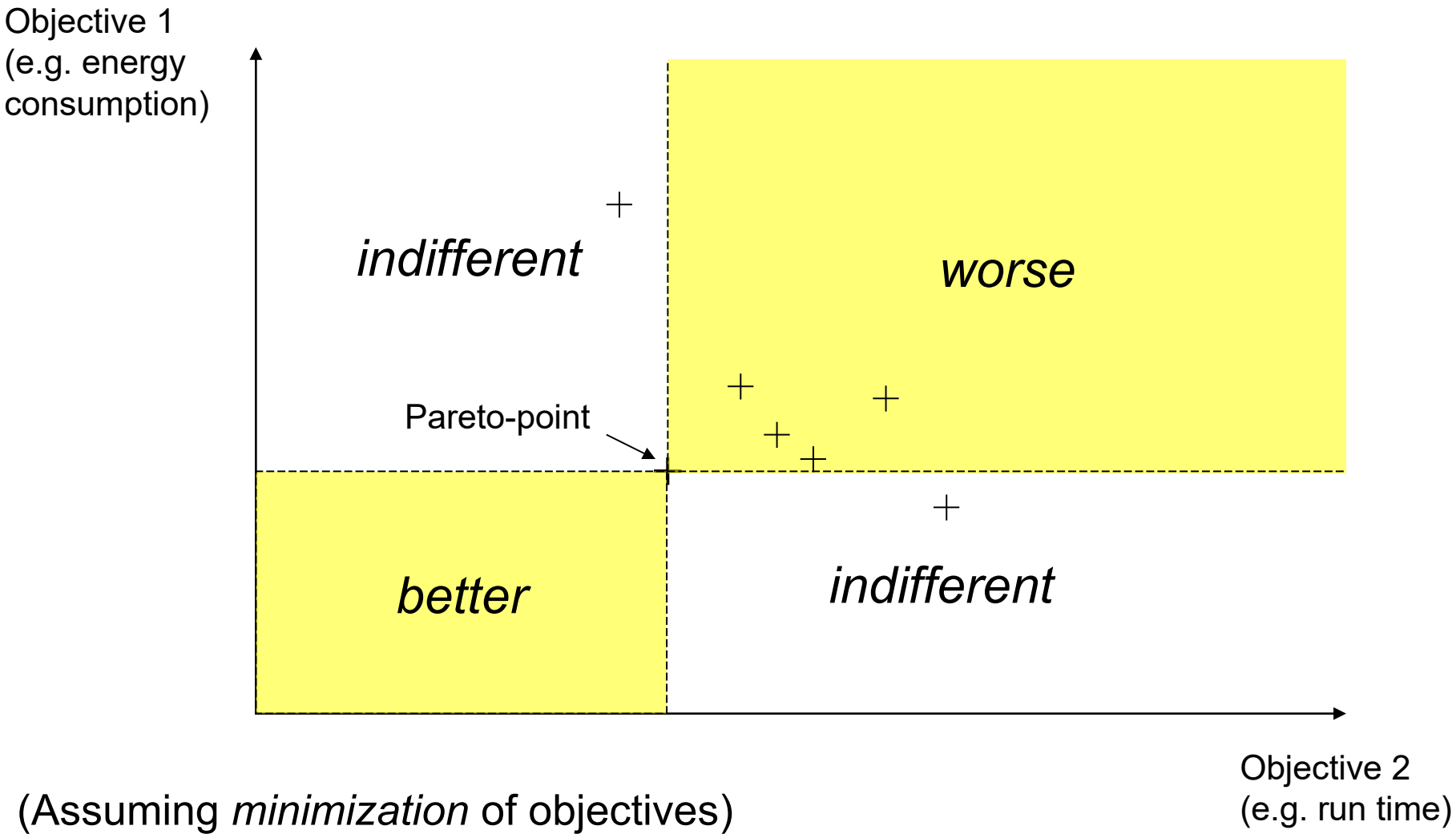
- **Definition:**
  Vector $u \in F$ is **indifferent** with respect to vector $v \in F$
  $\Leftrightarrow$ neither u dominates v nor v dominates u

Pareto points (2)

- A solution $x \in X$ is called **Pareto-optimal** with respect to X
  $\Leftrightarrow$ there is no solution $y \in X$ such that $u=f(x)$ is dominated by $v=f(y)$. x is a **Pareto point**.
- **Definition**: Let $S \subseteq F$ be a subset of solutions.
  $v \in F$ is called a **non-dominated solution** with respect to S
  $\Leftrightarrow$ v is not dominated by any element $\in S$.
- v is called **Pareto-optimal**
  $\Leftrightarrow$ v is non-dominated with respect to all solutions F.
- A **Pareto-set** is the set of all Pareto-optimal solutions

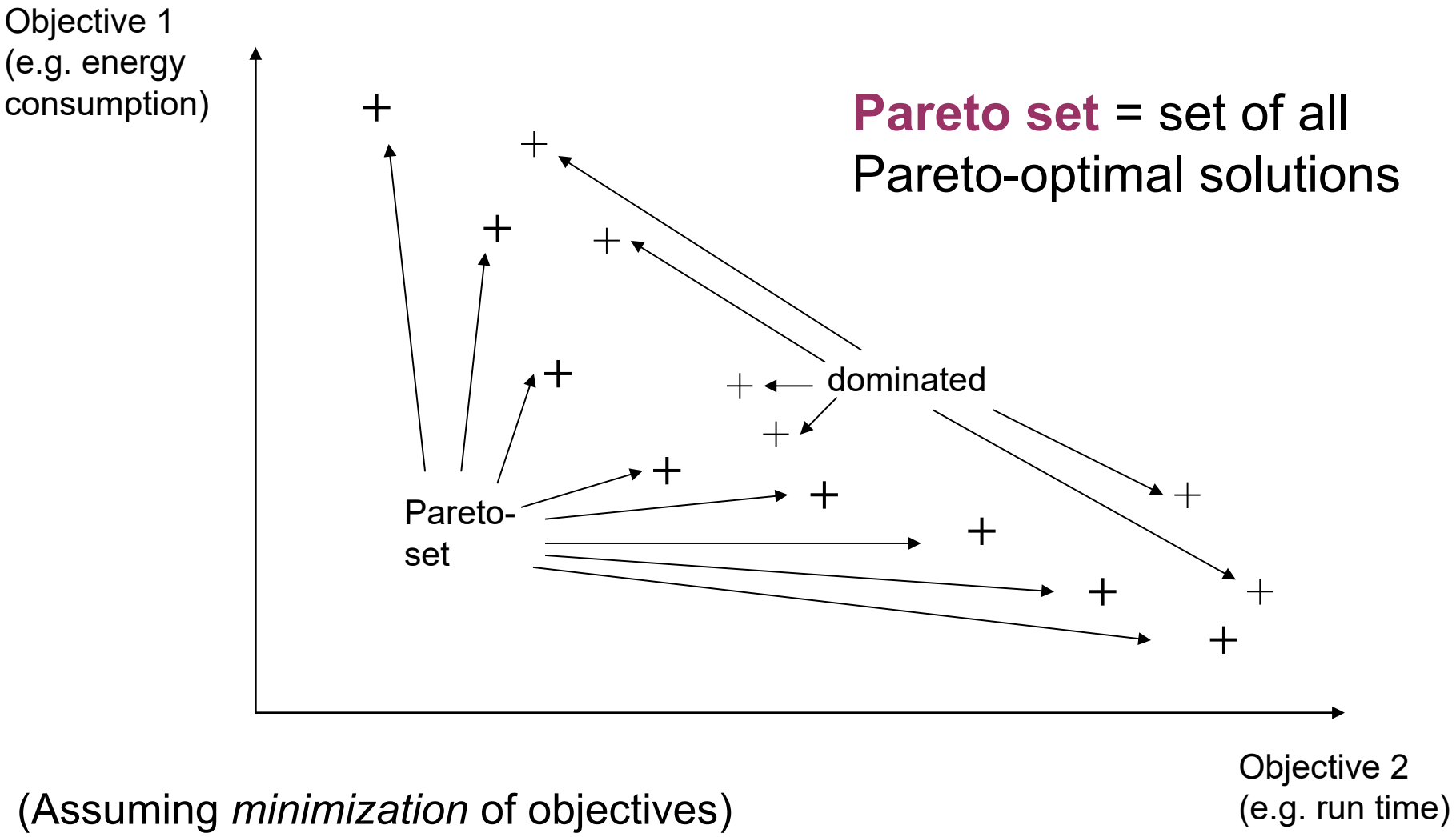
Pareto-sets define a **Pareto-front**
(boundary of dominated subspace)

Pareto Point



(Assuming *minimization* of objectives)

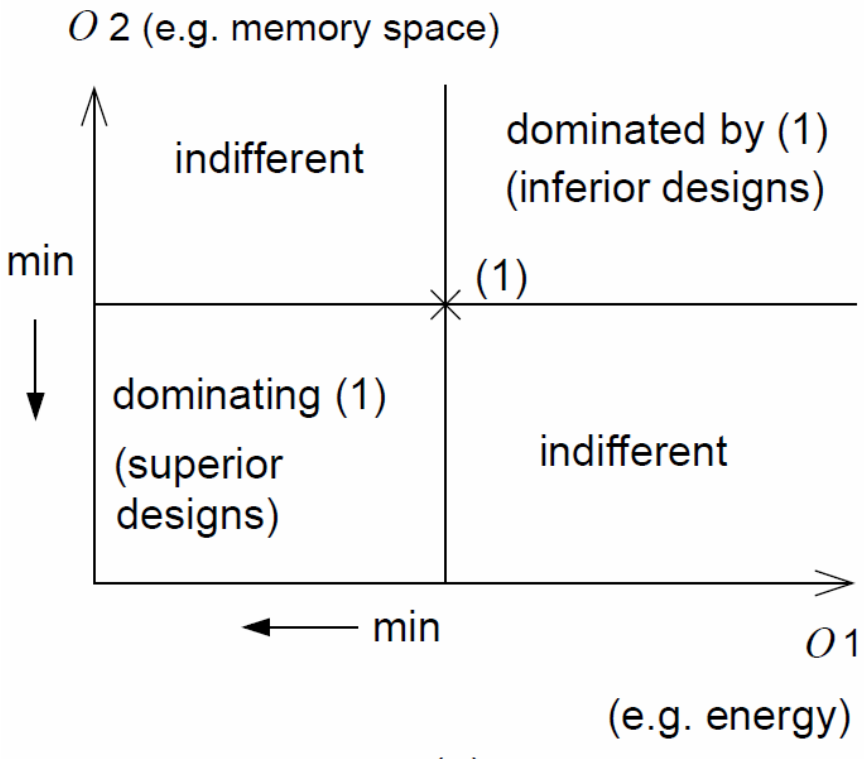
Pareto Set



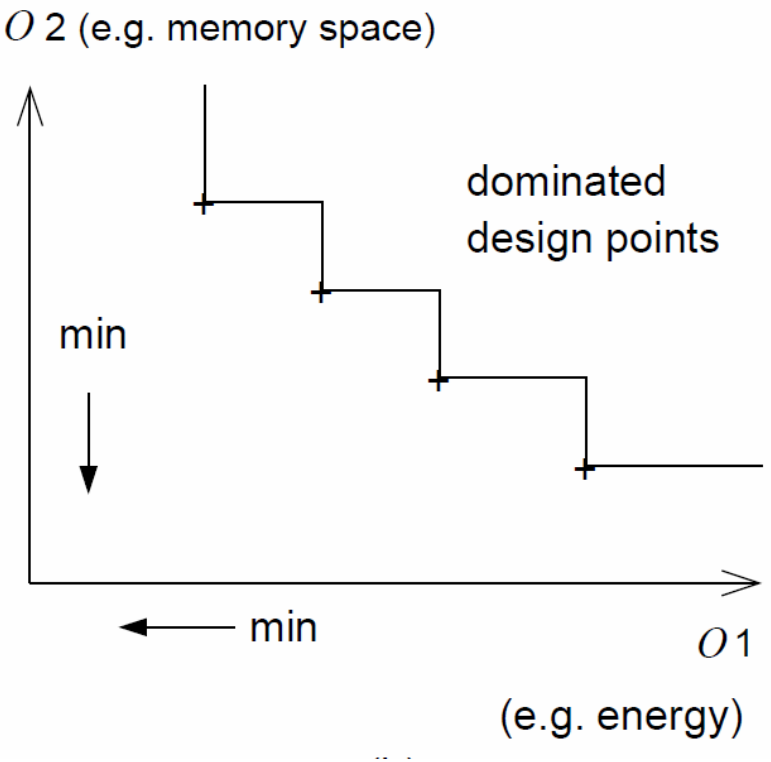
(Assuming *minimization* of objectives)

One more time …

Pareto point



Pareto front



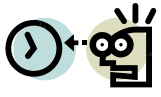
Design space evaluation

Design space evaluation (DSE) based on Pareto-points is the process of finding and returning a set of Pareto-optimal designs to the user, enabling the user to select the most appropriate design.

How to evaluate designs according to multiple criteria?

Many different criteria are relevant for evaluating designs:

- Average & worst case delay
- power/energy consumption
- thermal behavior
- reliability, safety, security
- cost, size
- weight
- EMC characteristics
- radiation hardness, environmental friendliness, ..





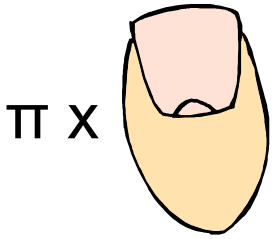




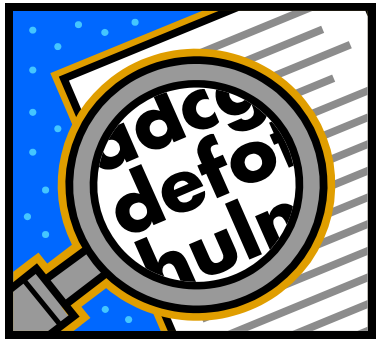
How to compare different designs?
(Some designs are “better” than others)

Average delays (execution times)

- **Estimated** average execution times **:**
  Difficult to generate sufficiently precise estimates;
  Balance between run-time and precision



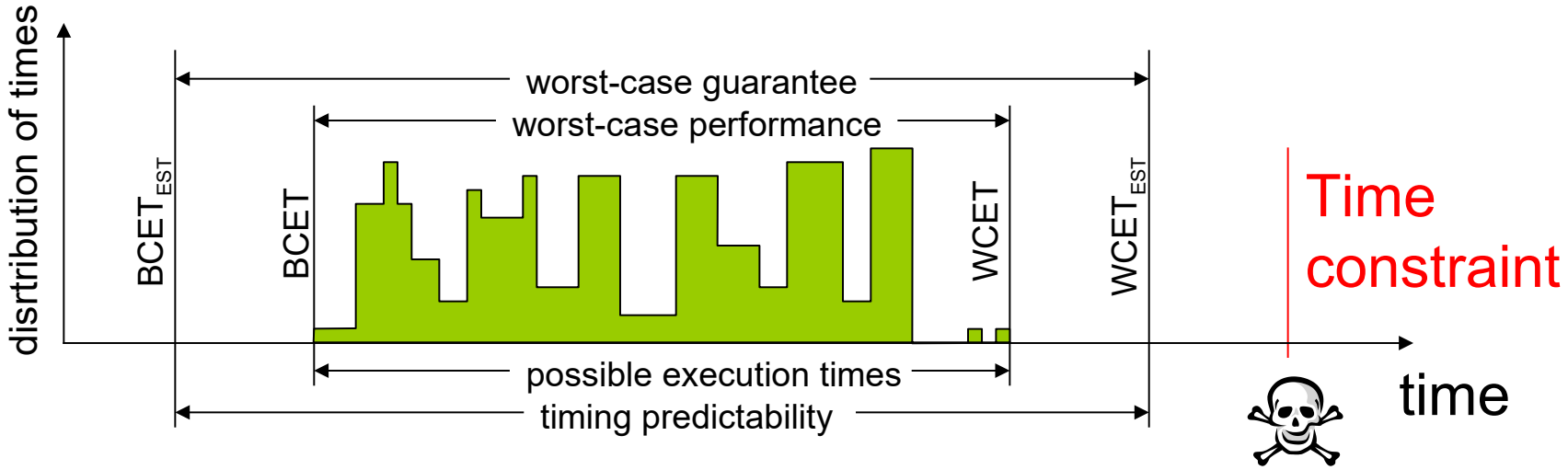
- **Accurate** average execution times**:**
  As precise as the input data is.



We need to compute **average** and **worst case** execution times

Worst case execution time (1)

Definition of worst case execution time:



$WCET_{EST}$ must be

1. safe (i.e. ≥ WCET) and
2. tight ($WCET_{EST}$-WCET ≪ $WCET_{EST}$)

Worst case execution time (2)

Complexity:

- in the general case: undecidable if a bound exists.
- for restricted programs: simple for “old“ architectures, very complex for new architectures with pipelines, caches, interrupts, virtual memory, etc.

Approaches:

- for hardware: requires detailed timing behavior
- for software: requires availability of machine programs; complex analysis (see, e.g., https://www.absint.com)

Einschub: Optimierungsalgorithmen

– Beginn –

technische universität
dortmund

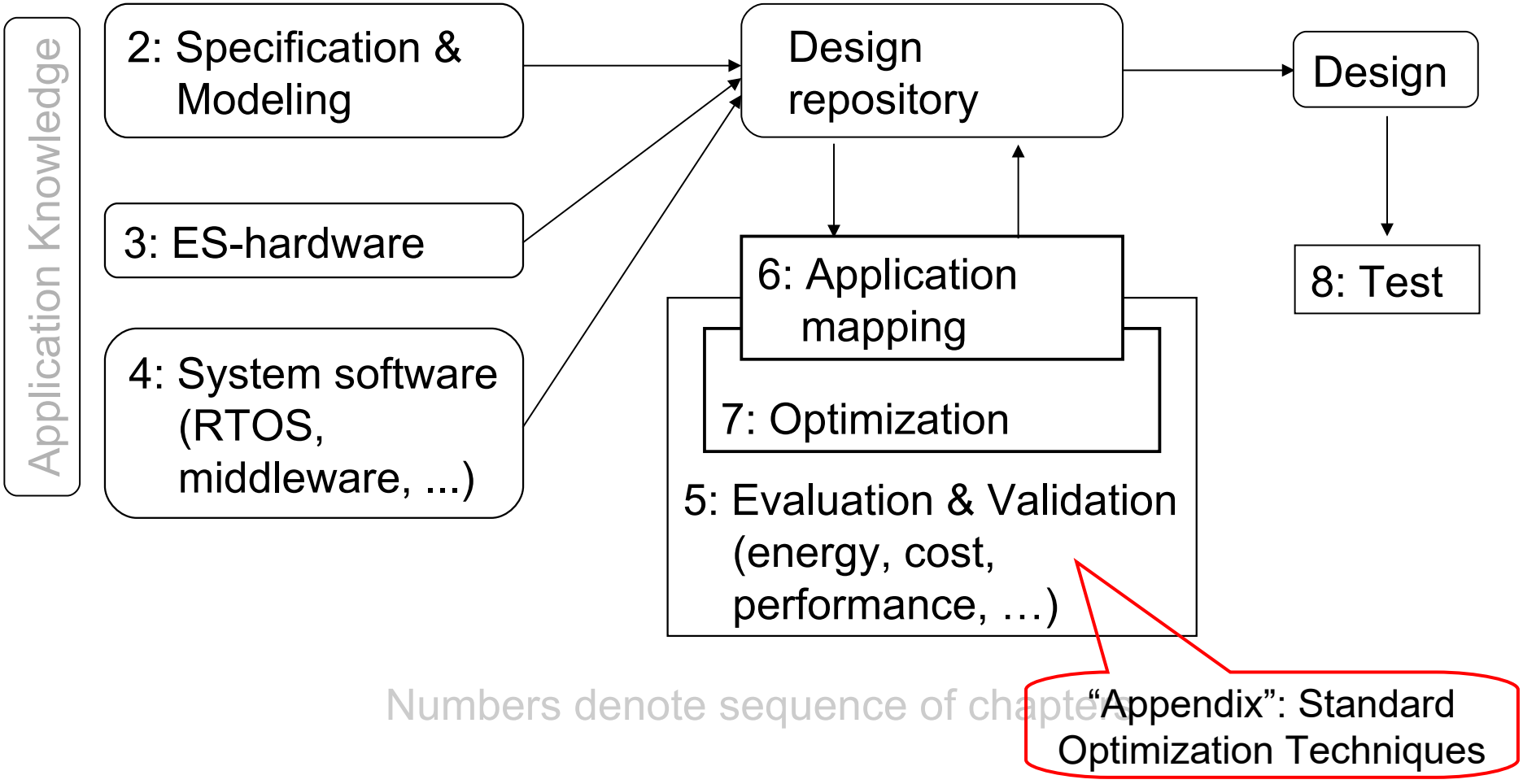
fakultät für informatik
informatik 12

Standard Optimization Techniques

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 12 月 19 日

Structure of this course



Integer linear programming models

Ingredients:

- Cost function
- Constraints

} Involving linear expressions of integer variables from a set X

Cost function

$$C = \sum_{x_i \in X} a_i x_i \text{ with } a_i \in \mathbb{R}, x_i \in \mathbb{N} \quad (1)$$

Constraints

$$\forall j \in J : \sum_{x_i \in X} b_{i,j} x_i \geq c_j \text{ with } b_{i,j}, c_j \in \mathbb{R} \quad (2)$$

Def.: The problem of minimizing (1) subject to the constraints (2) is called an **integer linear programming (ILP) problem**.

If all x_i are constrained to be either 0 or 1, the ILP problem is said to be a **0/1 integer linear programming problem**.

Example

$$C = 5x_1 + 6x_2 + 4x_3$$

$$x_1 + x_2 + x_3 \geq 2$$
$$x_1, x_2, x_3 \in \{0, 1\}$$

x_1	x_2	x_3	C	
0	1	1	10	
1	0	1	9	← Optimal
1	1	0	11	
1	1	1	15	

Remarks on integer programming

- Maximizing the cost function: just set $C' = -C$
- Integer programming is NP-complete.
- Running times depend exponentially on problem size, but problems of >1000 vars solvable with good solver (depending on the size and structure of the problem)
- The case of $x_i \in \mathbb{R}$ is called *linear programming* (LP). Polynomial complexity, but most algorithms are exponential, in practice still faster than for ILP problems.
- The case of some $x_i \in \mathbb{R}$ and some $x_i \in \mathbb{N}$ is called *mixed integer-linear programming.*
- ILP/LP models good starting point for modeling, even if heuristics are used in the end.
- Solvers: lp_solve (public), CPLEX (commercial), …

Simulated Annealing

- General method for solving combinatorial optimization problems.
- Based the model of slowly cooling crystal liquids.
- Some configuration is subject to changes.
- Special property of Simulated annealing: Changes leading to a poorer configuration (with respect to some cost function) are accepted with a certain probability.
- This probability is controlled by a temperature parameter: the probability is smaller for smaller temperatures.

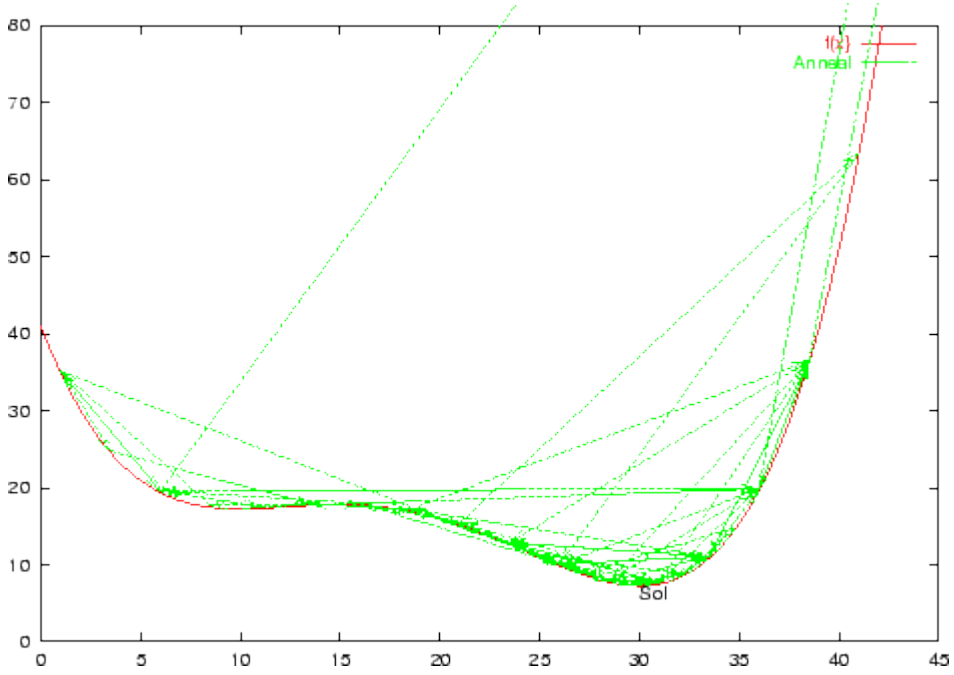
Simulated Annealing Algorithm

```
procedure SimulatedAnnealing;
  var i, T: integer;
begin
  i := 0; T := MaxT;
  configuration:= <some initial configuration>;
  while not terminate(i, T) do
  begin
    while InnerLoop do
    begin NewConfig := variation(configuration);
      delta := evaluation(NewConfig,configuration);
      if delta < 0
      then configuration := NewConfig;
      else if SmallEnough(delta, T, random(0,1))
        then configuration := NewConfig;
    end;
    T:= NewT(i,T); i:=i+1;
  end; end;
```

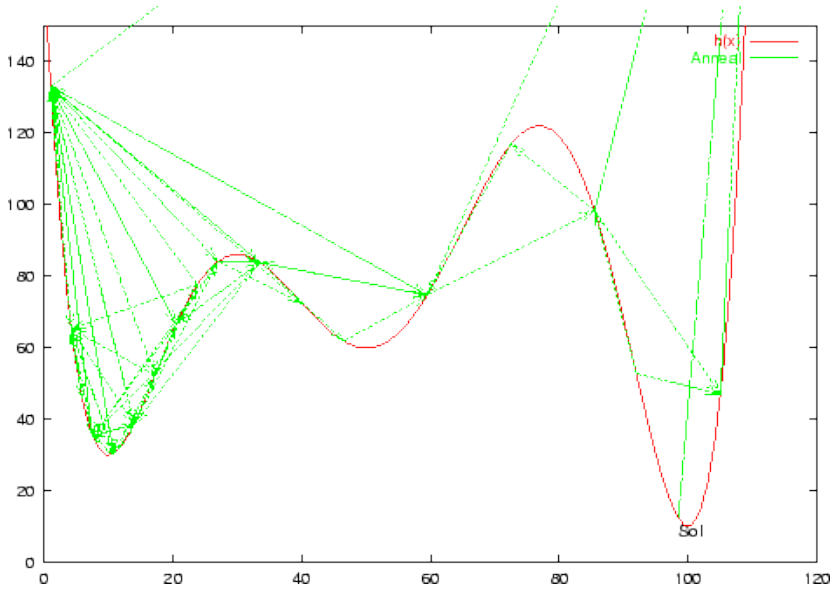
Explanation

- Initially, some random initial configuration is created.
- Current temperature is set to a large value.
- Outer loop:
  - Temperature is reduced for each iteration
  - Terminated if (temperature $\leq$ lower limit) or (number of iterations $\geq$ upper limit).
- Inner loop: For each iteration:
  - New configuration generated from current configuration
  - Accepted if (new cost $\leq$ cost of current configuration)
  - Accepted with temperature-dependent probability if (cost of new config. > cost of current configuration).

Behavior for actual functions



130 steps



200 steps

e.g.: https://cadapplets.lafayette.edu/fp/fpIntro.html

Performance

- This class of algorithms has been shown to outperform others in certain cases [Wegener, 2005].
- Demonstrated its excellent results in the TimberWolf layout generation package [Sechen]
- Many other applications …

Evolutionary Algorithms (1)

- ***Evolutionary Algorithms*** *are based on the collective learning process within a population of individuals, each of which represents a search point in the space of potential solutions to a given problem.*
- *The population is arbitrarily initialized, and it evolves towards better and better regions of the search space by means of randomized processes of*
  - ***selection*** *(which is deterministic in some algorithms),*
  - ***mutation****, and*
  - ***recombination*** *(which is completely omitted in some algorithmic realizations).*

Bäck, Schwefel, 1993

Evolutionary Algorithms (2)

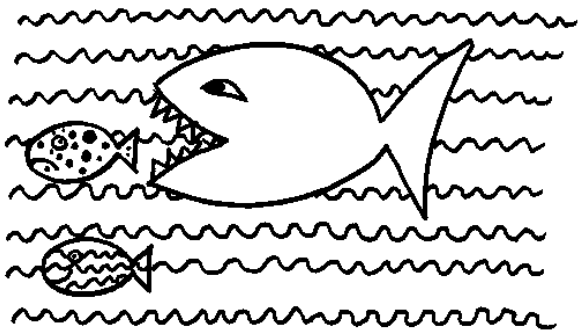
- *The environment (given aim of the search) delivers a quality information (**fitness value**) of the search points, and the selection process favours those individuals of higher fitness to reproduce more often than worse individuals.*
- *The recombination mechanism allows the mixing of parental information while passing it to their descendants, and mutation introduces innovation into the population.*

Bäck, Schwefel, 1993

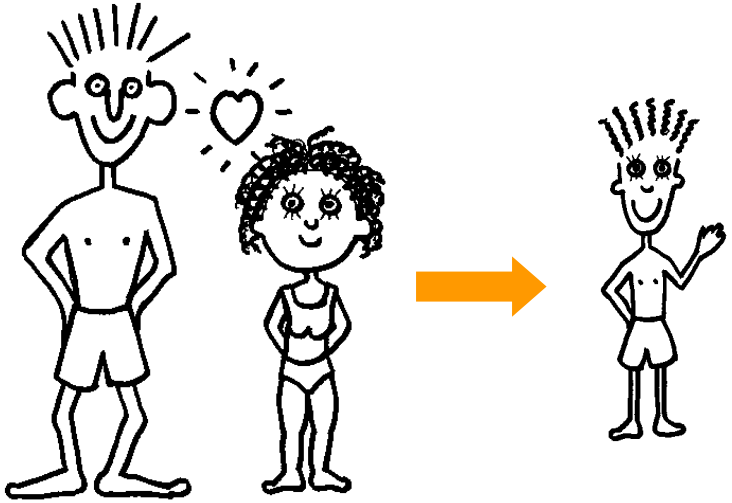
Evolutionary Algorithms (3)

Principles of Evolution

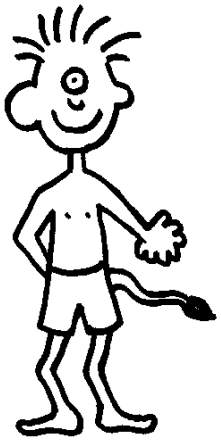
❶ Selection



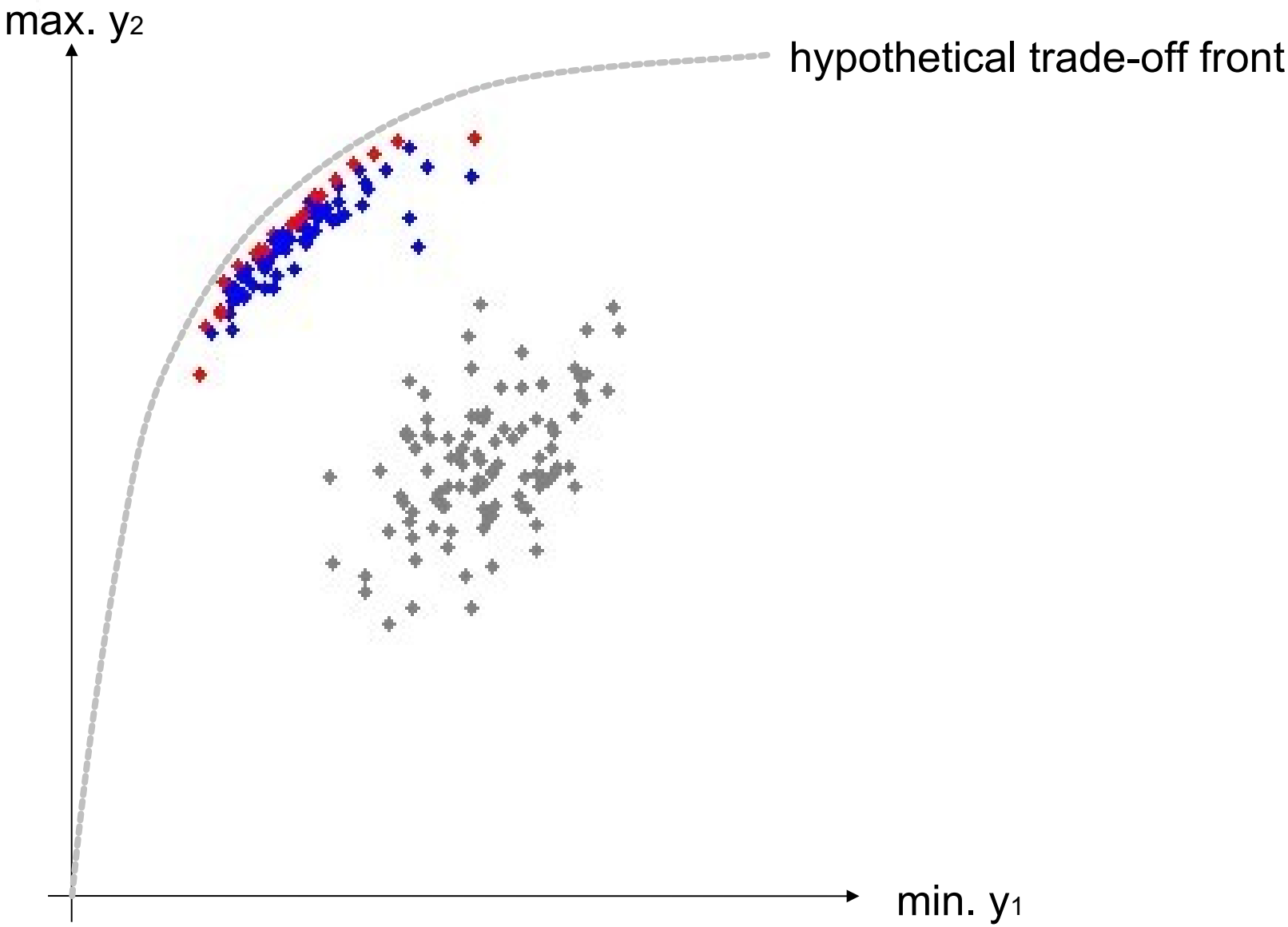
❸ Cross-over



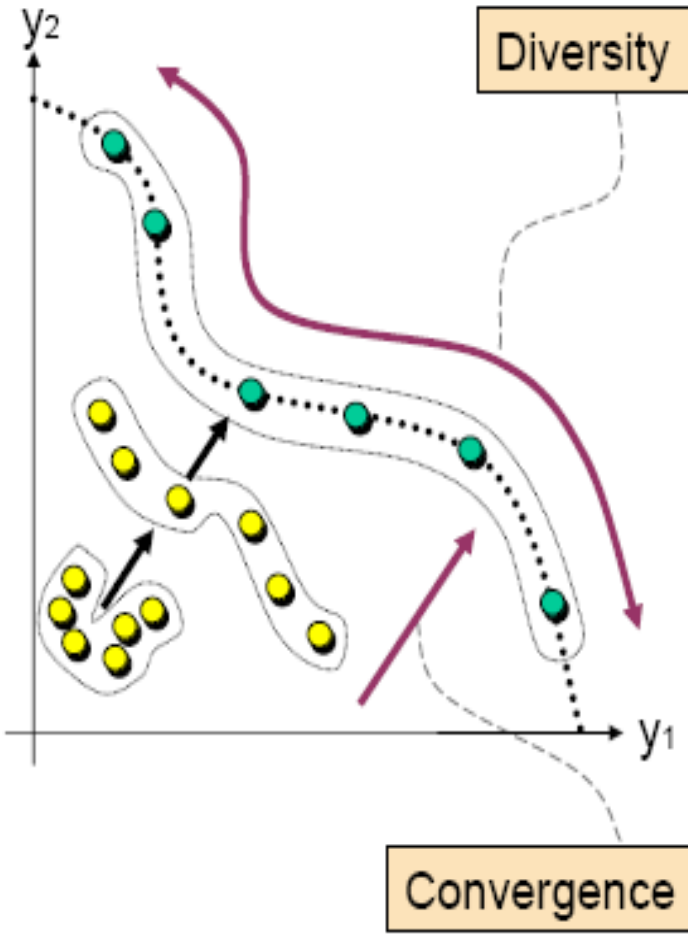
❷ Mutation



An Evolutionary Algorithm in Action



Issues in Multi-Objective Optimization



- How to maintain a diverse Pareto set approximation?

  ❷ **density estimation**

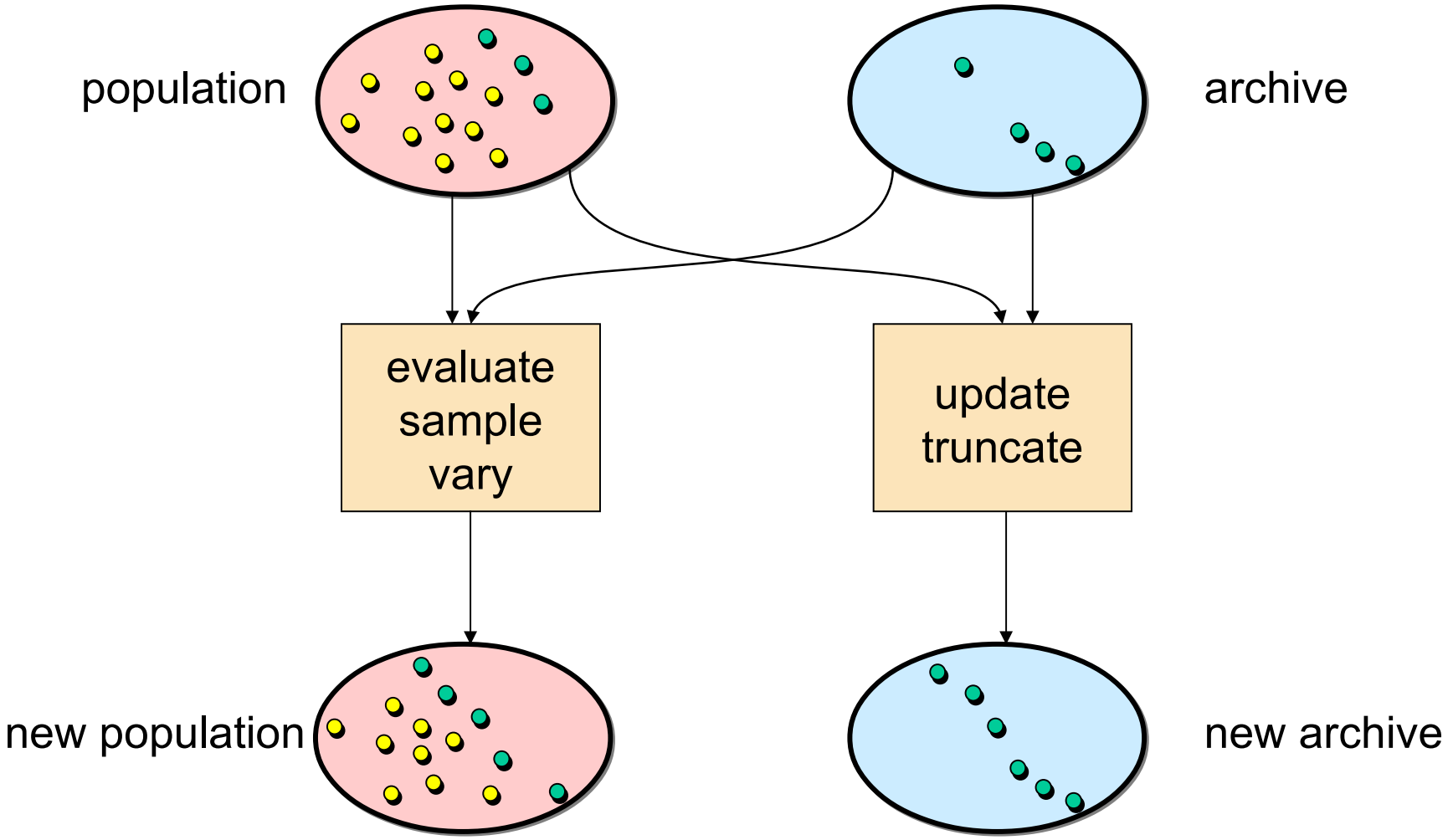
- How to prevent nondominated solutions from being lost?

  ❸ **environmental selection**

- How to guide the population towards the Pareto set?

  ❶ **fitness assignment**

A Generic Multiobjective EA



Example: SPEA2 Algorithm

Step 1:	Generate initial population P0 and empty archive (external set) A_0. Set t = 0.
Step 2:	Calculate fitness values of individuals in P_t and A_t.
Step 3:	A_{t+1} = nondominated individuals in P_t and A_t. If size of A_{t+1} > N then reduce A_{t+1}, else if size of A_{t+1} < N then fill A_{t+1} with dominated individuals in P_t and A_t.
Step 4:	If t > T then output the nondominated set of A_{t+1}. Stop.
Step 5:	Fill mating pool by binary tournament selection.
Step 6:	Apply recombination and mutation operators to the mating pool and set P_{t+1} to the resulting population. Set t = t + 1 and go to Step 2.

Summary

Integer (linear) programming

- Integer programming is NP-complete
- Linear programming is faster
- Good starting point even if solutions are generated with different techniques

Simulated annealing

- Modeled after cooling of liquids
- Overcomes local minima

Evolutionary algorithms

- Maintain set of solutions
- Include selection, mutation and recombination

Einschub: Optimierungsalgorithmen

– Ende –

– Beginn –

What does Execution Time Depend on

- Input parameters
  - Algorithm parameters
  - Problem size
  - etc.
- Initial states and intermediate states of the system for executing
  - Cache configuration, replacement policies
  - Pipelines
  - Speculations
  - etc.
- Interferences from the environment
  - Scheduling
  - Interrupts
  - etc.

How to Derive the Worst-Case Execution Time (WCET)

- Most of industry's best practice
  - Measure it: determine WCET directly by running or simulating a set of inputs.
    - There is no guarantee to give an upper bound of the WCET.
    - The derived WCET could be too optimistic.
  - Exhaustive execution: by considering the set of all the possible inputs
    - In general, not possible
    - The inputs have to cover all the possible initial states and intermediate states of the system, which is also usually not possible.
- Compute it
  - In general, not possible neither, as computing (tight) WCET for a program is *uncomputable* by Turing machines.
  - Based on some structures, it is possible and the derived solution is a safe upper bound of the WCET.

Why is It Uncomputable?

Halting Problem

Given the description of a Turing machine m and its input x, the problem is to answer whether the machine halts on x.

Theorem

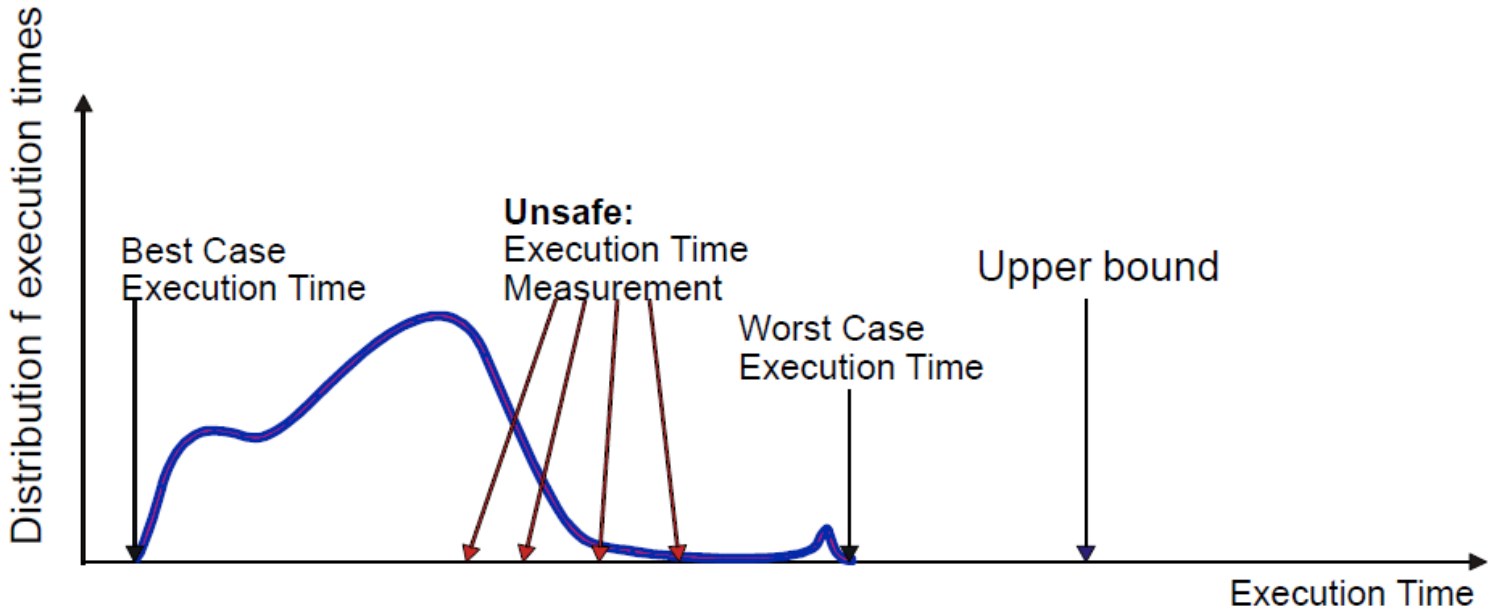
Halting Problem is undecidable (uncomputable). In other words, one cannot use an algorithm to decide whether another algorithm m halts on a specific input.

WCET is undecidable

It is even undecidable if it terminates at all. Deriving the WCET is of course of undecidable.

Please refer to the textbook of Computational Complexity by Prof. Papadimitriou.

Execution Time Distribution



Our objectives:

- *Upper bound* of execution time as tightly as possible.
- All control-flow paths, by considering all possible inputs.
- All paths through the architecture, resulting from the potential initial and assumed intermediate architectural states.

Timing Analysis

By considering systems, in general, with

- finite architectural configurations, finite input domains, and bounded loops and recursion,

WCET is computable.

But....., the search space is too large to explore it exhaustively!

Why is It Hard for Analyzing WCET?

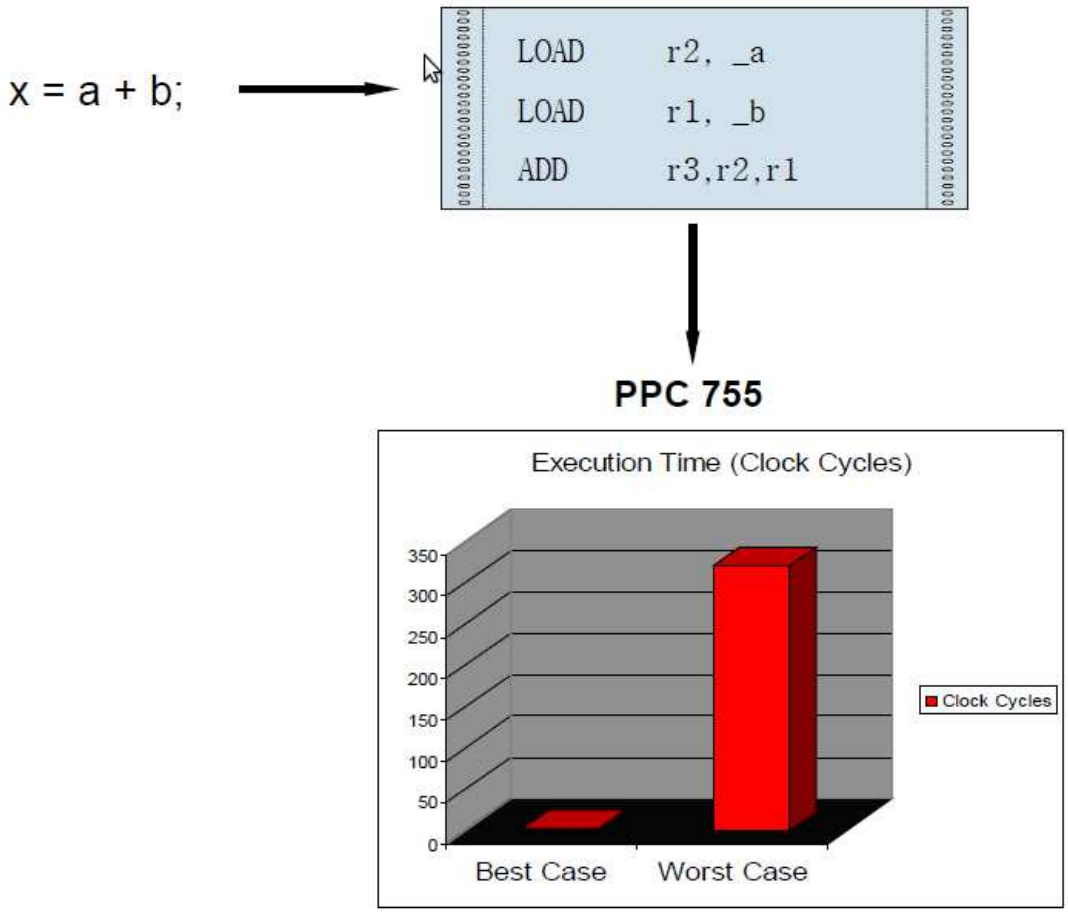
Execution time $e(i)$ of machine instruction i

- In the good old time:
  $e(i)$ is a constant c, which could be found in the data sheet
- Nowadays, especially for high-performance processors:
  $e(i)$ also depends on the (architectural) execution state s.

$$\min\{e(i,s)|s \in S\} \leq e(i) \leq \max\{e(i,s)|s \in S\},$$

  where S is the set of all states.
- Using $\max\{e(i,s)|s \in S\}$ is safe for WCET, but might be not tight since some states in S might not be possible reached by some inputs.
- Execution history, resulting in a smaller set of reachable execution states, has to be enforced to improve the tightness of the analysis.

Variability of Execution Times



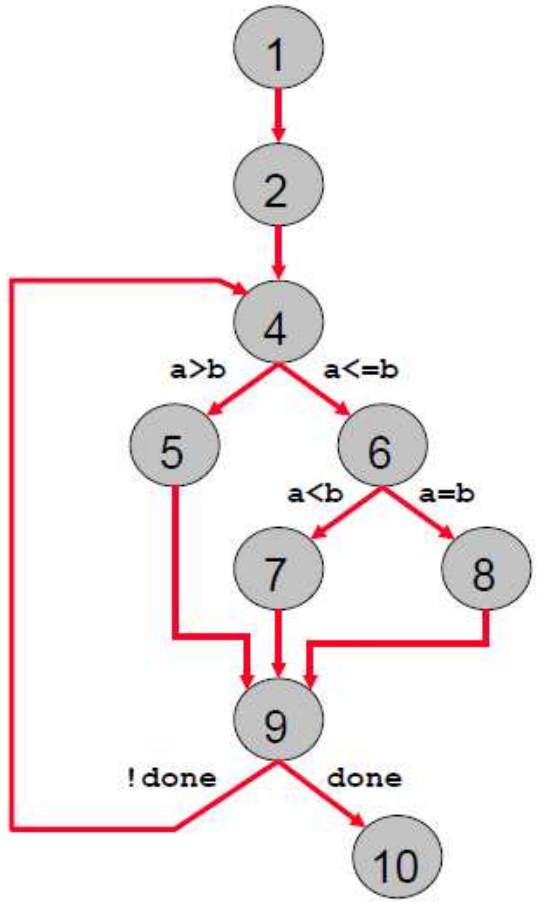
Wilhelm et al.

Timing Accidents and Penalties

- Timing Accident: cause for an increase of the execution time of an instruction
- Timing Penalty: the associated increase
- Types of timing accidents
  - Cache misses
  - Pipeline stalls
  - Branch mispredictions
  - Bus collisions
  - Memory refresh of DRAM
  - TLB miss

Control Flow Graph (CFG)

```
   what_is_this {
1     read (a,b);
2     done = FALSE;
3     repeat {
4       if (a>b)
5          a = a-b;
6       elseif (b>a)
7          b = b-a;
8       else done = TRUE;
9     } until done;
10    write (a);
   }
```



Basic Blocks

Definition: *A basic block is a sequence of instructions where the control flow enters at the beginning and exits at the end, in which it is highly amenable to analysis.*

Determining the basic blocks

$$a[0] := b[0] + c[0]$$
$$a[1] := b[3] + c[3]$$
$$a[2] := b[6] + c[6]$$
$$d := a[0] * a[1]$$
$$e := d / a[2]$$
$$\text{if } e < 10 \text{ goto } L$$

- Beginning:
  - the first instruction
  - targets of un/conditional jumps
  - instructions that follow un/conditional jumps
- Ending:
  - the basic block consists of the block beginning and runs until the next block beginning (exclusive) or until the program ends

Program Path Analysis

- Problem: Which sequence of instructions is executed in the worst case (i.e., the longest execution time)?
- Input:
  - Timing information for each basic block, derived from static analysis (value/cache/pipeline analysis)
  - Loop bounds by specification
  - CFG derived from the executable binary program
- Basic Concept:
  - Transform structure of CFG into a set of (integer) linear equations
  - Solution of the Integer Linear Program (ILP) yields bound on the WCET.

Program Path Analysis: Formal Definition

Input

A CFG with N basic blocks, in which each basic block B_i has a worst-case execution time c_i, given by static analysis.

Output

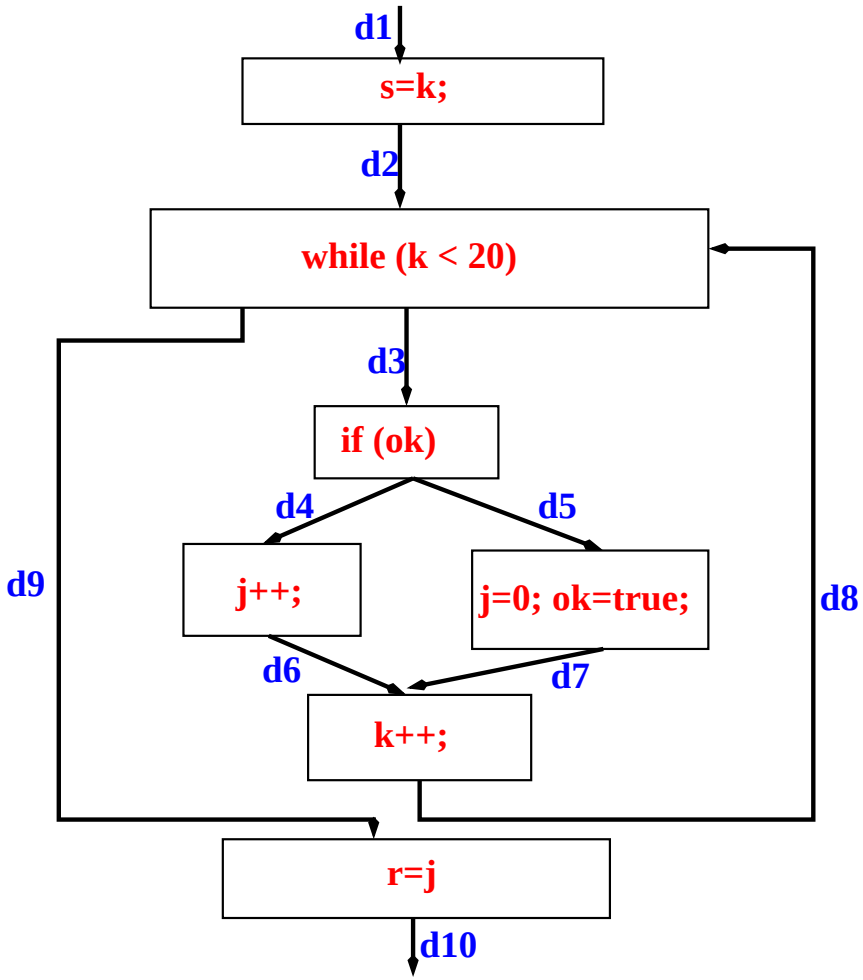
Suppose that each block B_i is executed *exactly* x_i times. What is the worst-case execution time

$$WCET = \sum_{i=1}^{N} c_i \cdot x_i,$$

such that the values of x_is satisfy the structural constraints in the CFG?

Note that additional constraints provided by the programmer (bounds for loop counters, etc.) can also be included.

Example for CFG Constraints



Flow equations: (x_i is a variable)

$d_1 = d_2 = x_1$

$d_3 + d_9 = d_2 + d_8 = x_2$

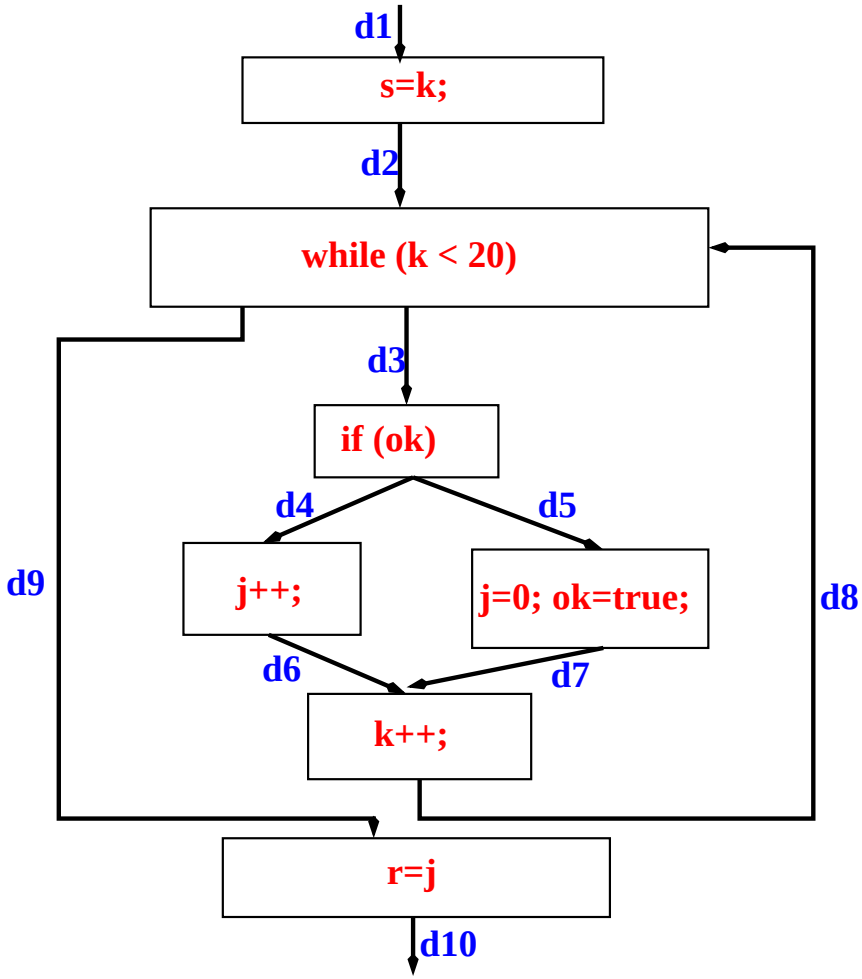
$d_4 + d_5 = d_3 = x_3$

$d_6 + d_7 = d_8 = x_4$

$d_4 = d_6 = x_5$

$d_5 = d_7 = x_6$

Example for Additional Constraints



The loop is executed for at most 20 times when k is initialized with a non-negative number:

$$x_3 \leq 20x_1.$$

The basic block for $j = 0; ok = true;$ is executed for at most one time:

$$x_6 \leq x_1.$$

WCET: ILP Formulation

$$\text{maximize} \quad \sum_{i=1}^{N} c_i \cdot x_i$$

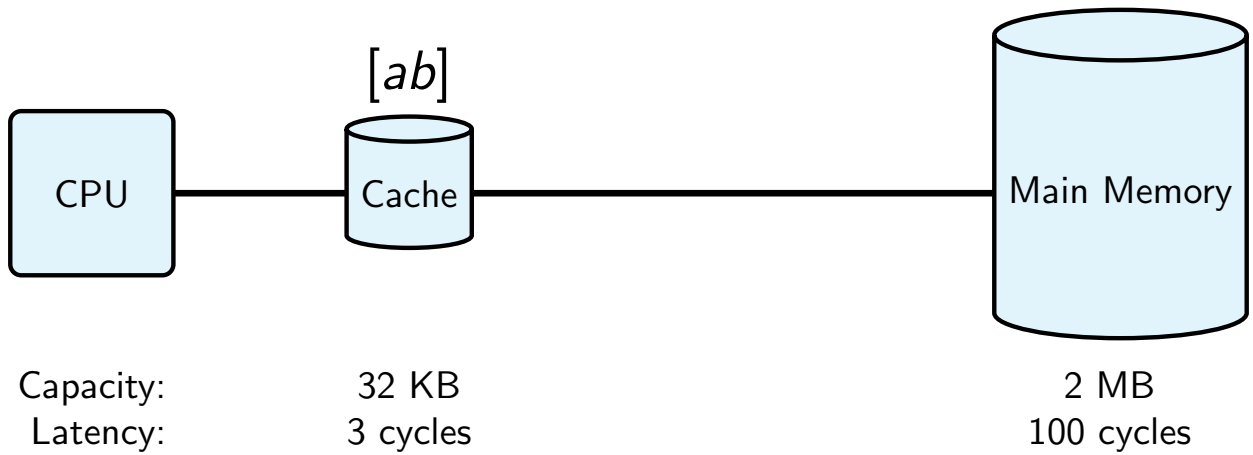
$$\text{such that} \quad d_1 = 1$$

$$\sum_{j \in in(B_i)} d_j = \sum_{k \in out(B_j)} d_k = x_i, \qquad \forall i = 1, \ldots, N$$

additional linear constraints

Caches: Fast Memory to Deal with the Memory Wall

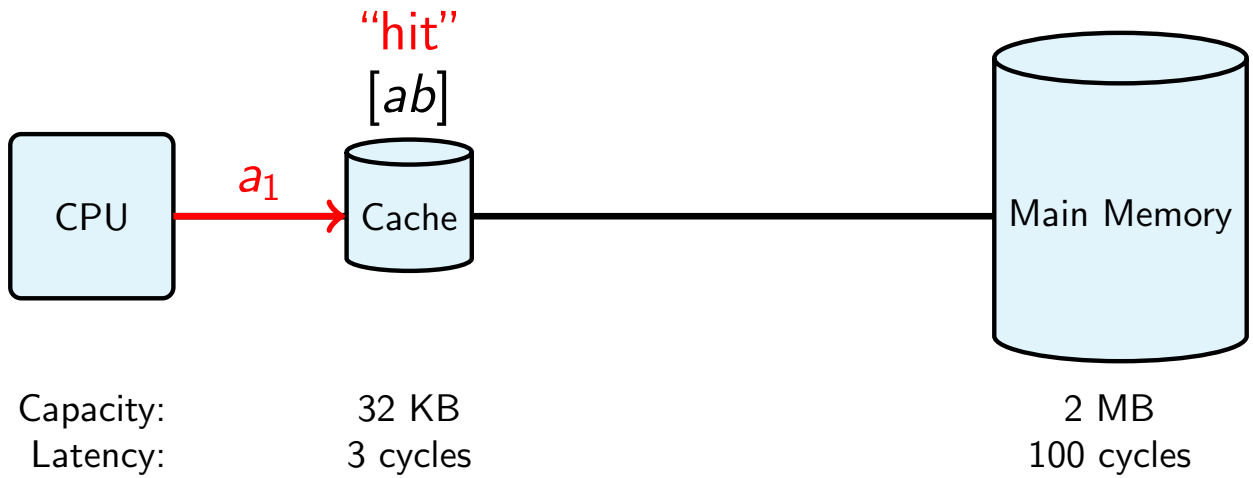
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

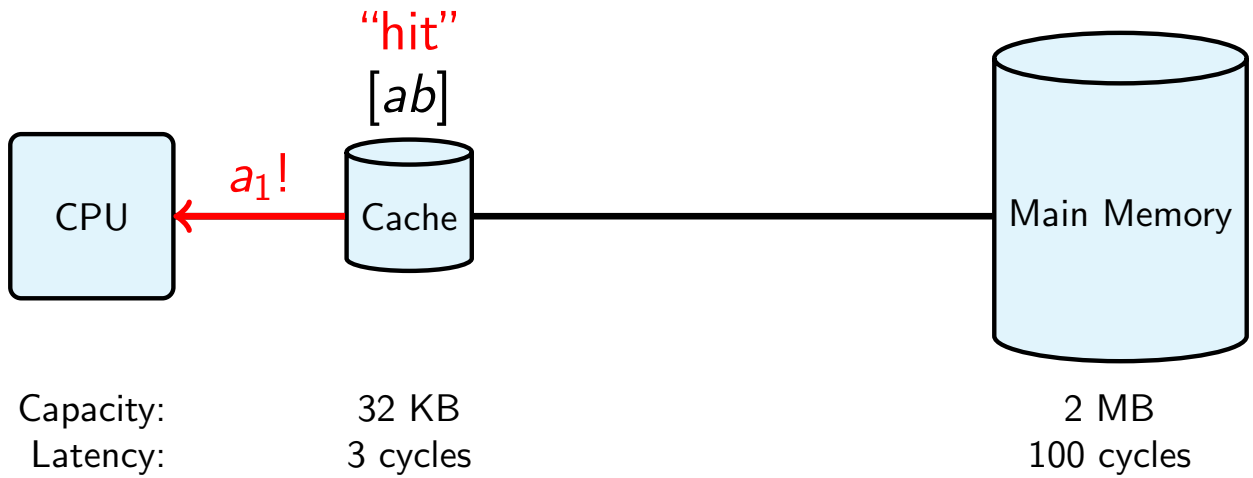
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

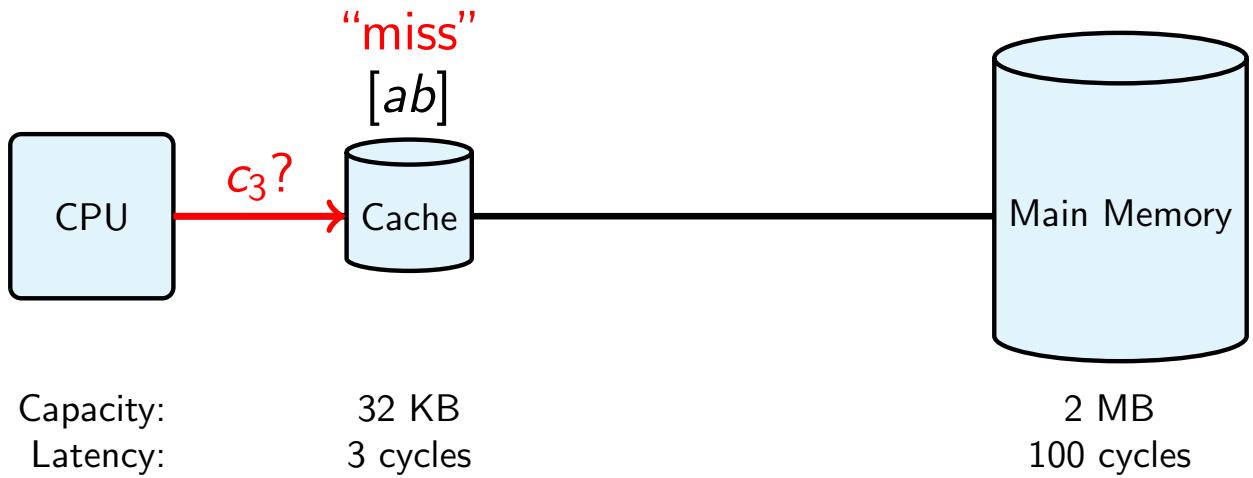
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

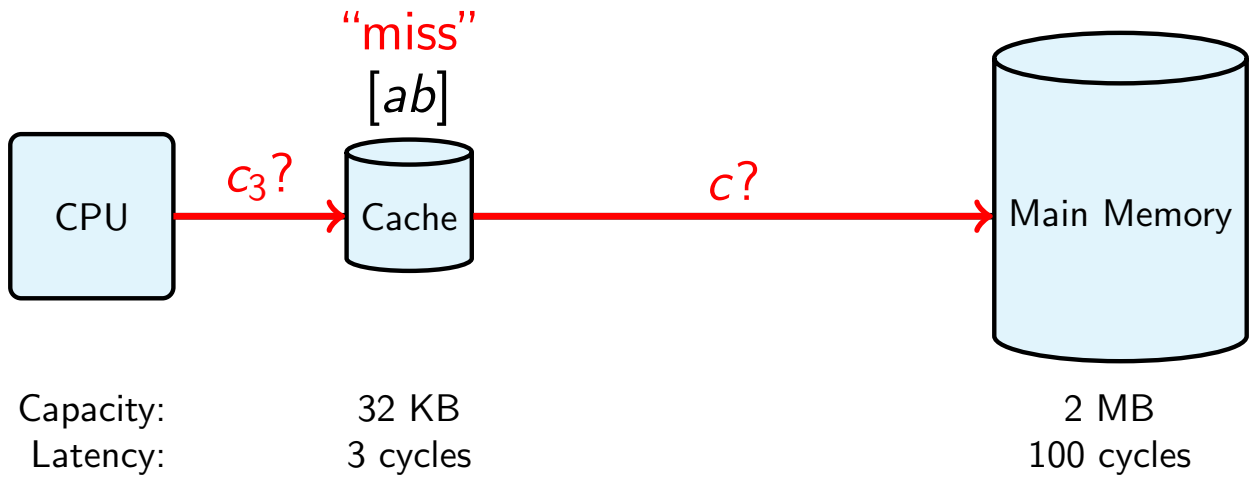
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

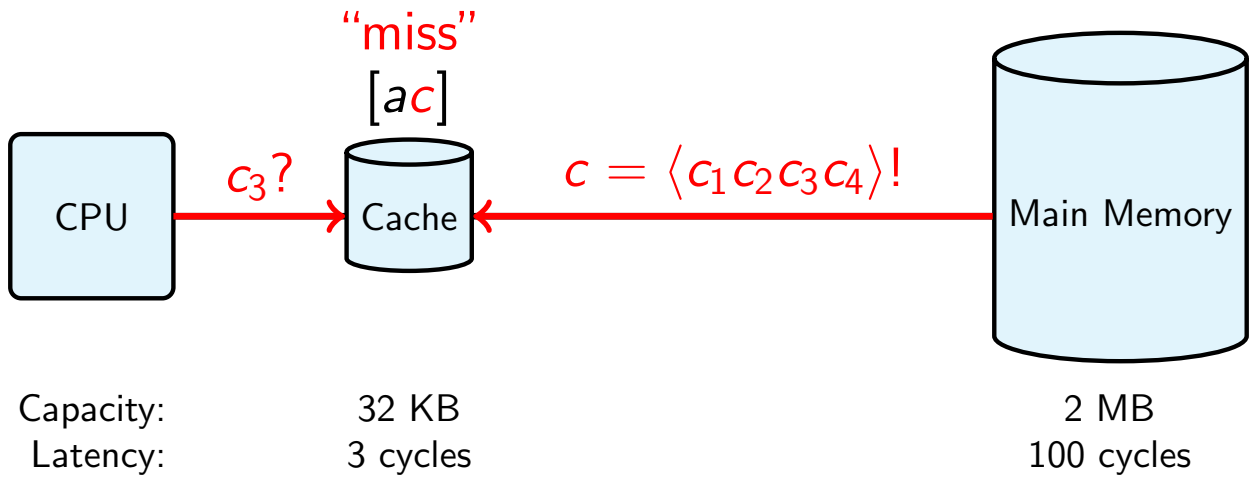
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

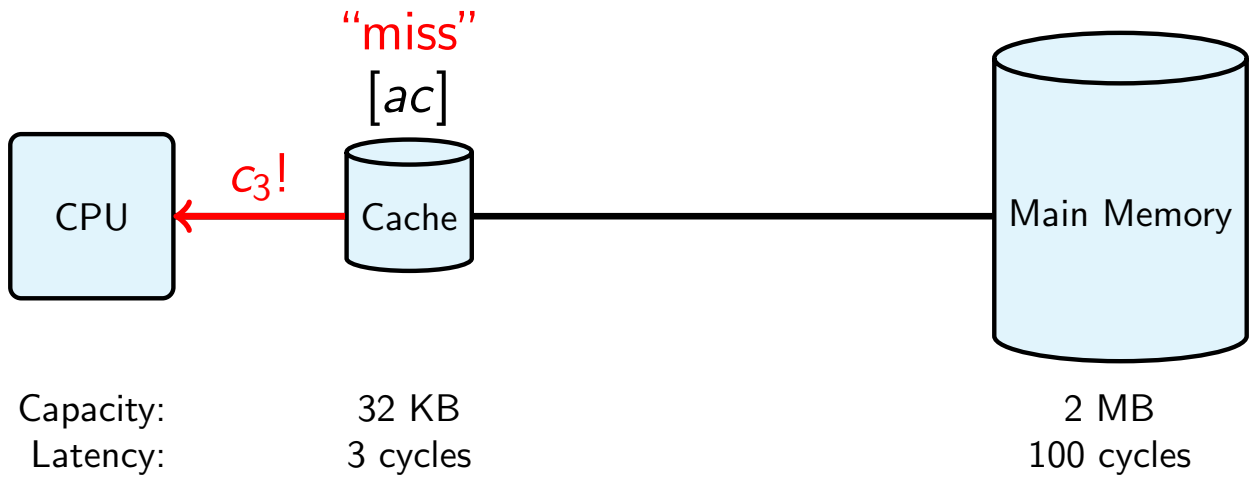
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

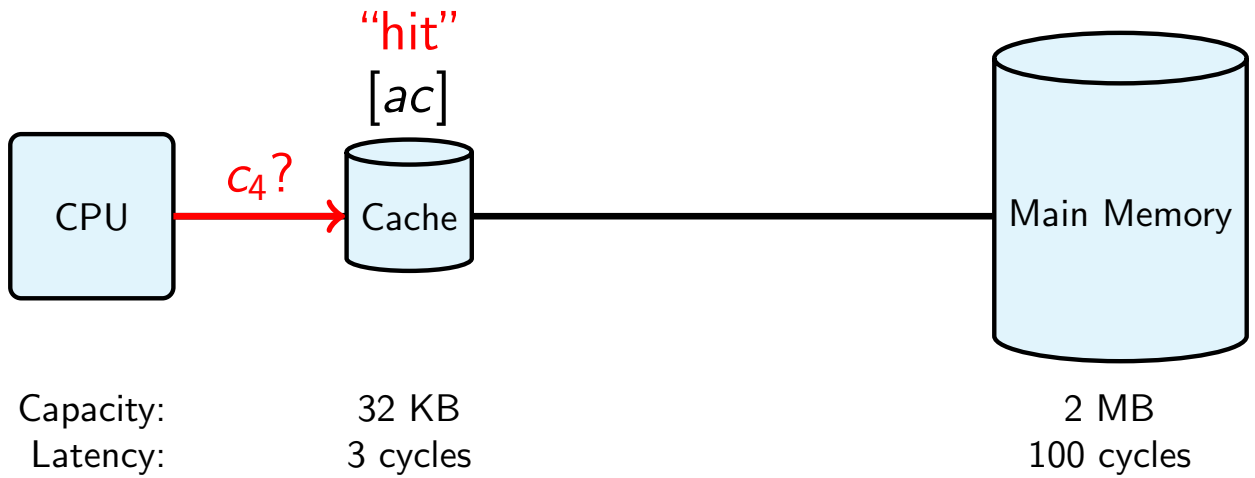
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

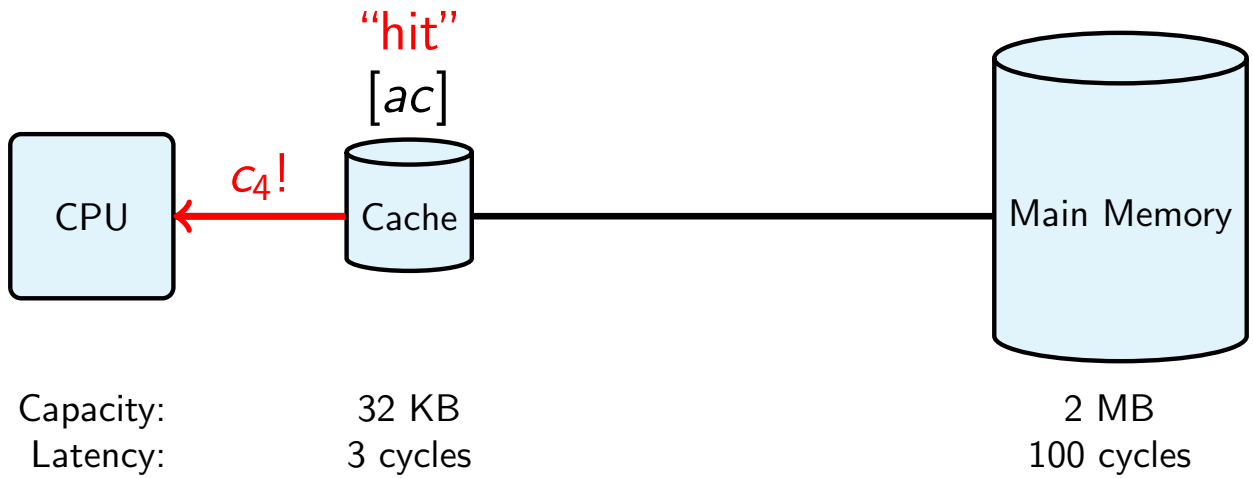
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

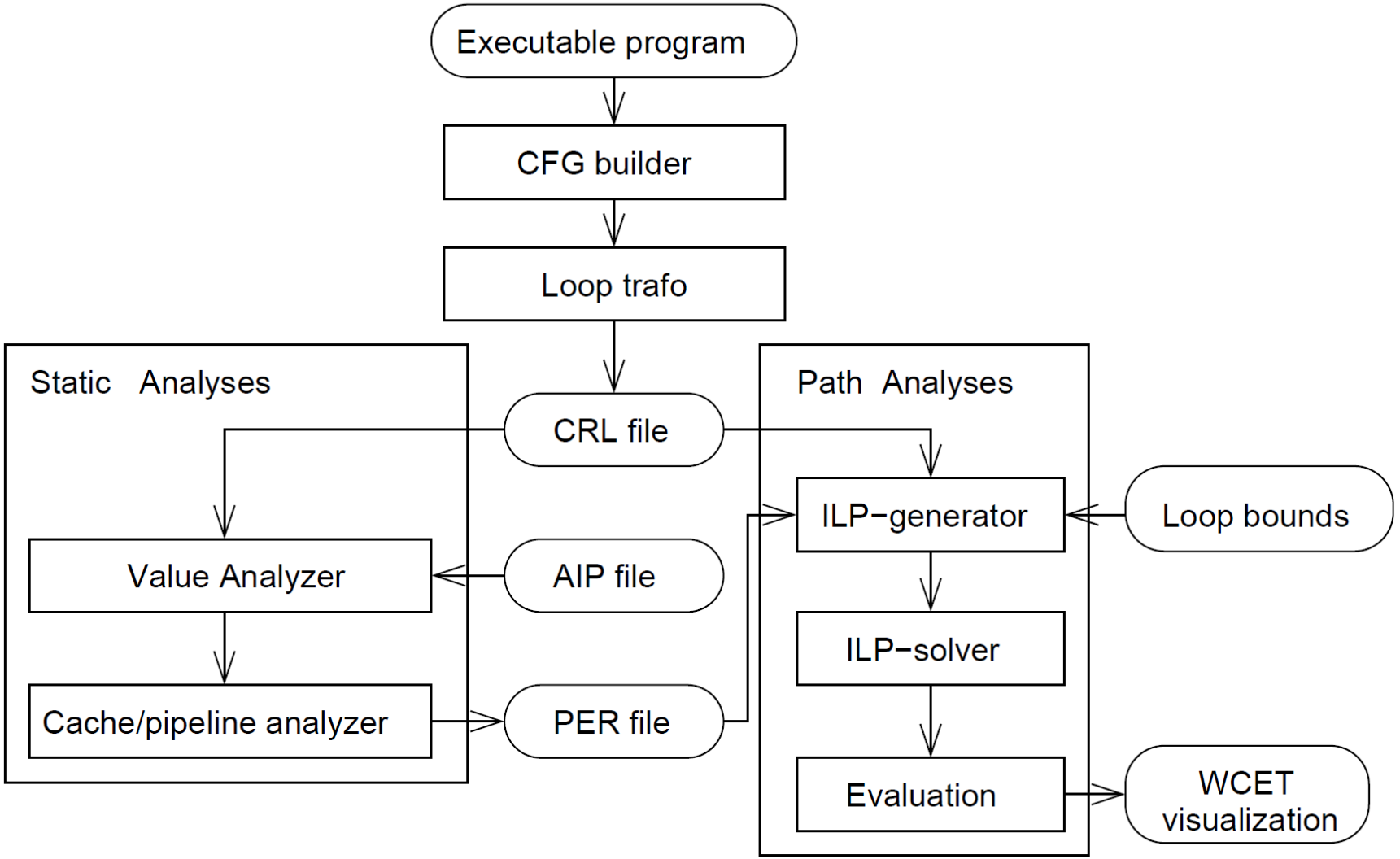
- How they work:
  - dynamically
  - managed by replacement policy



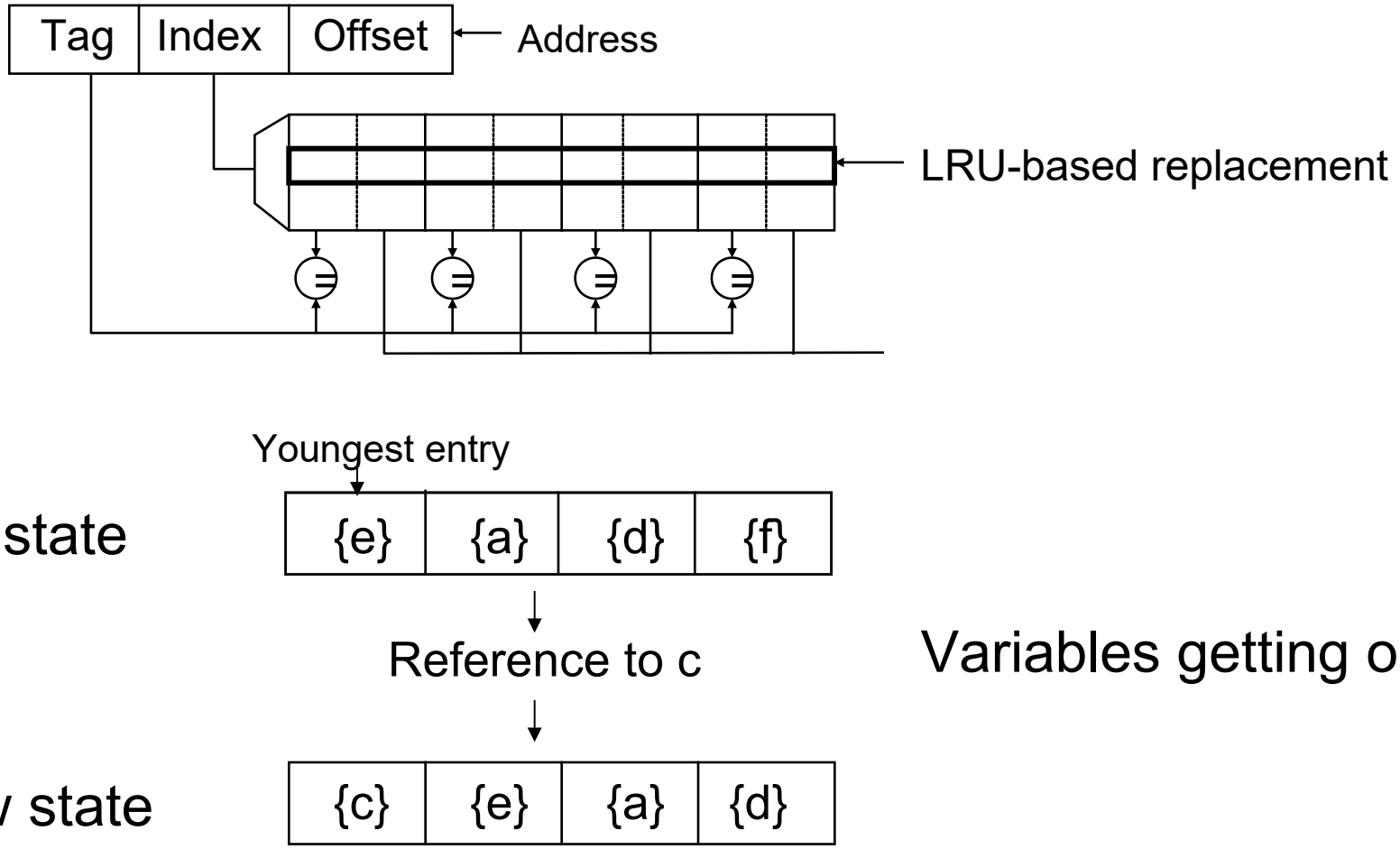
- Why they work: *principle of locality*
  - spatial
  - temporal

– Ende –

WCET estimation: AiT (AbsInt)

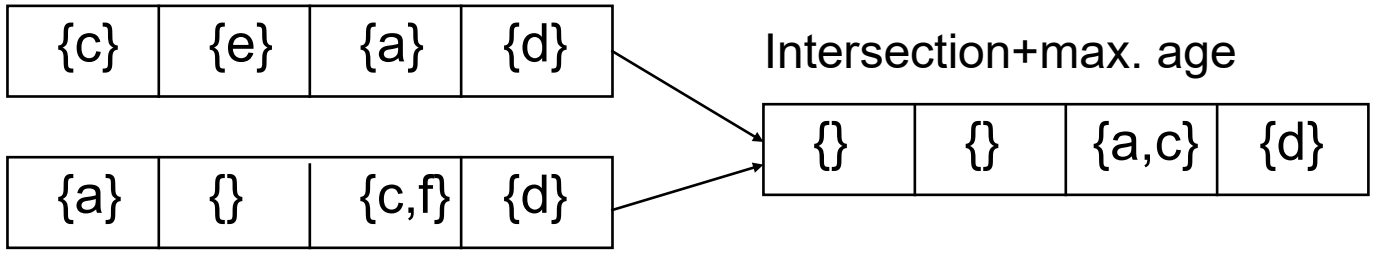


WCET estimation for caches

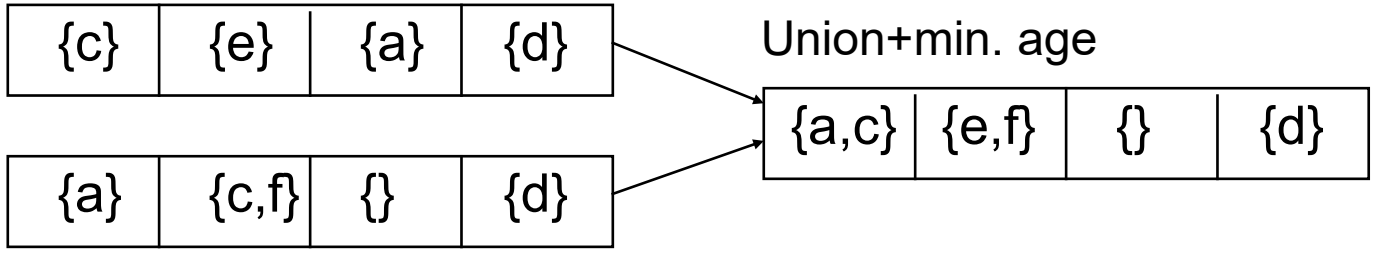


Behavior at program joins

Worst case

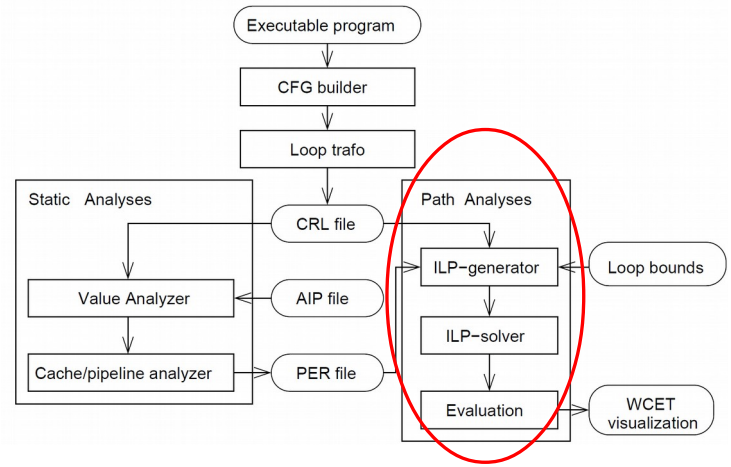


Best case



☞ Possibly several variables per entry

ILP model



- Objective function reflects execution time as a function of the execution time of blocks. To be **maximized**.
- Constraints reflect dependencies between blocks.
- Avoids explicit consideration of all paths

☞ Called **implicit path enumeration** technique.

Example (1)

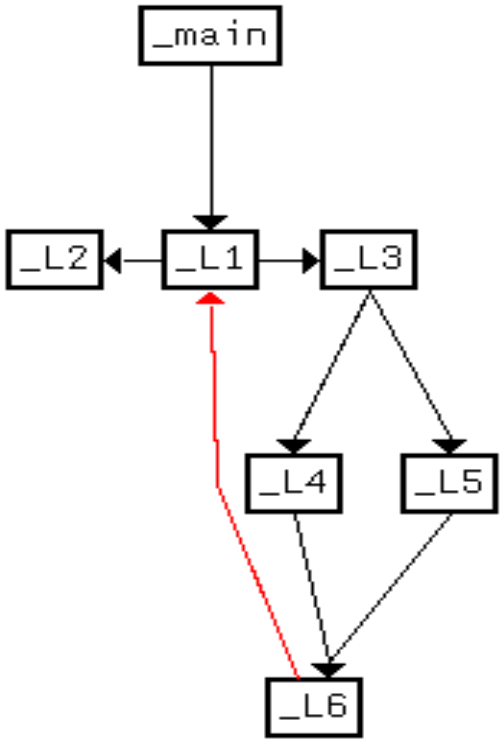
Program

```
int main()
{
  int i, j = 0;

  _Pragma( "loopbound min
                  100 max 100" );
  for ( i = 0; i < 100; i++ )
  { if ( i < 50 )
      j += i;
    else
      j += ( i * 13 ) % 42;
  }

  return j;
}
```

CFG



WCETs of BB (aiT 4 TriCore)

_main: 21 cycles
_L1: 27
_L3: 2
_L4: 2
_L5: 20
_L6: 13
_L2: 20

Example (2)

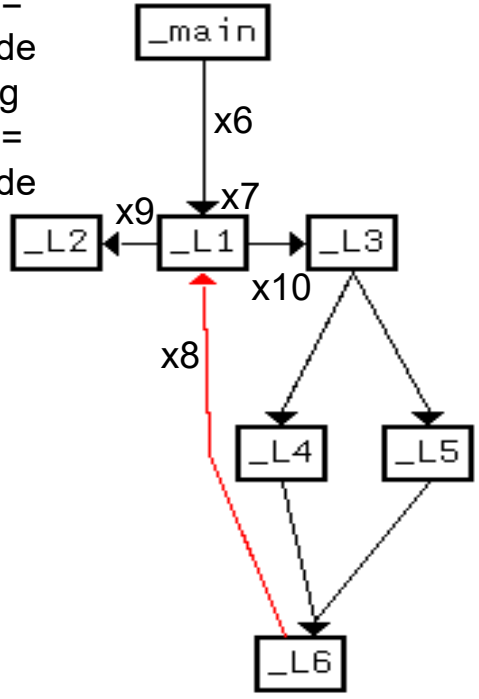
- Virtual start node
- Virtual end node
- Virtual end node per function

Variables:

- 1 variable per node
- 1 variable per edge

Constraints: „Kirchhoff“ equations per node

- Sum of incoming edge variables = flux through node
- Sum of outgoing edge variables = flux through node



_main: 21 cycles
_L1: 27
_L3: 2
_L4: 2
_L5: 20
_L6: 13
_L2: 20

ILP

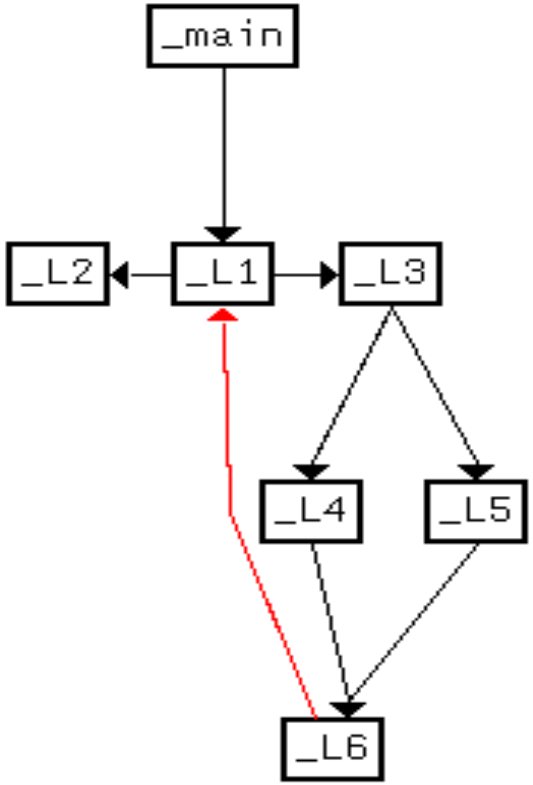
```
/* Objective function = WCET to be maximized*/
21 x2 + 27 x7 + 2 x11 + 2 x14 + 20 x16 + 13 x18 + 20 x19;
/* CFG Start Constraint */ x0 - x4 = 0;
/* CFG Exit Constraint */  x1 - x5 = 0;
/* Constraint for flow entering function main */
x2 - x4 = 0;
/* Constraint for flow leaving exit node of main */
x3 - x5 = 0;
/* Constraint for flow entering exit node of main */
x3 - x20 = 0;
/* Constraint for flow entering main = flow leaving main */
x2 - x3 = 0;
/* Constraint for flow leaving CFG node _main */
x2 - x6 = 0;
/* Constraint for flow entering CFG node _L1 */
x7 - x8 - x6 = 0;
/* Constraint for flow leaving CFG node _L1 */
x7 - x9 - x10 = 0;
/* Constraint for lower loop bound of _L1 */
x7 - 101 x9 >= 0;
/* Constraint for upper loop bound of _L1 */
x7 - 101 x9 <= 0; ….
```

Example (3)

Value of objective function: 6268

Actual values of the variables:

x2	1
x7	101
x11	100
x14	0
x16	100
x18	100
x19	1
x0	1
x4	1
x1	1
x5	1
x3	1
x20	1
x6	1
x8	100
x9	1
x10	100
x12	100
x13	0
x15	0
x17	100



Summary

Evaluation and Validation

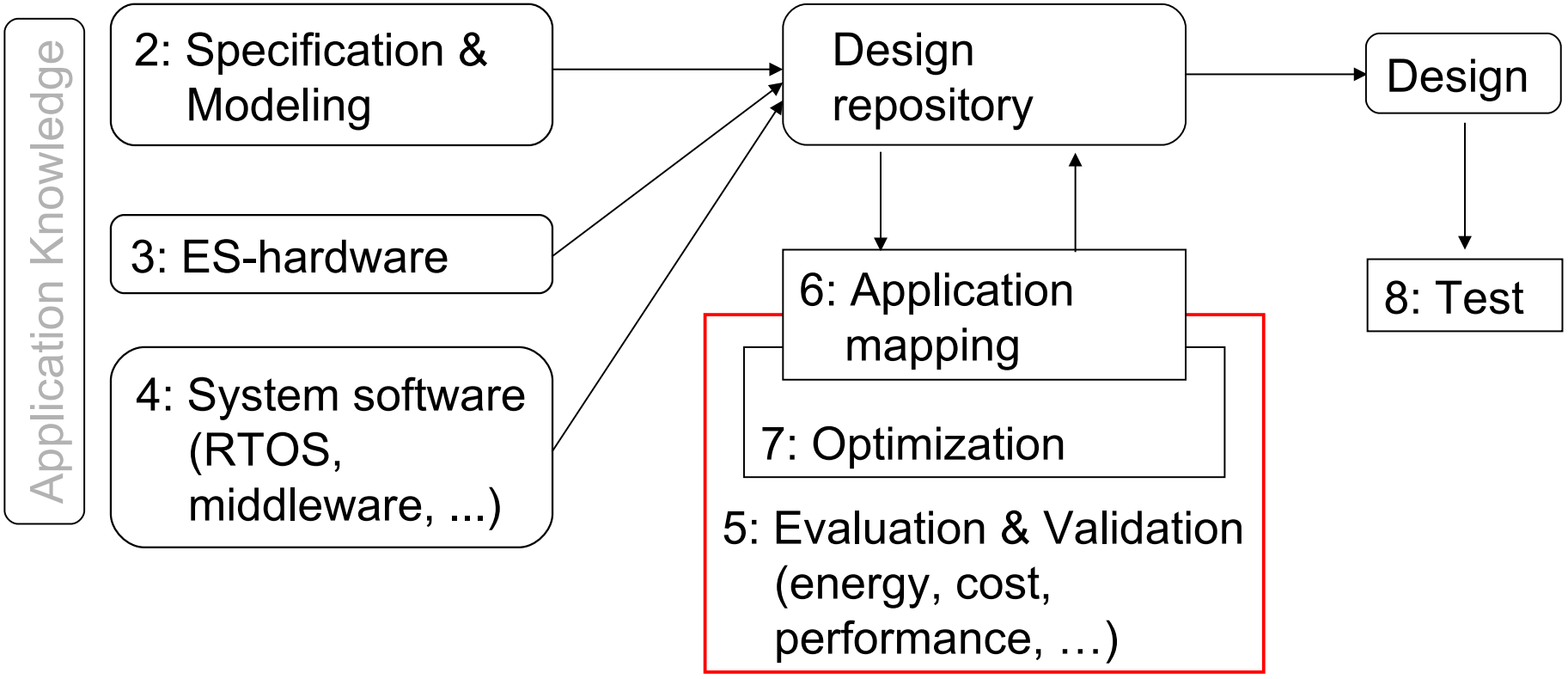
- In general, multiple objectives
- Pareto optimality
- Design space evaluation (DSE)
- Execution time analysis
  - Trade-off between speed and accuracy
  - Computation of worst case execution times
    - Cache/pipeline analysis
    - ILP model for computing WCET of application from WCET of blocks

Evaluation and Validation

Peter Marwedel
TU Dortmund,
Informatik 12

2013 年 12 月 03 日

Structure of this course

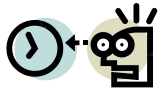


Numbers denote sequence of chapters

How to evaluate designs according to multiple criteria?

Many different criteria are relevant for evaluating designs:

- Average & worst case delay
- power/energy consumption
- thermal behavior
- reliability, safety, security
- cost, size
- weight
- EMC characteristics
- radiation hardness, environmental friendliness, ..







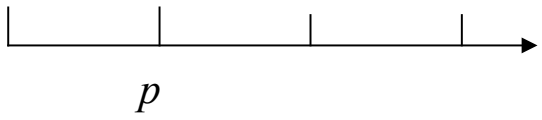


How to compare different designs?
(Some designs are “better” than others)

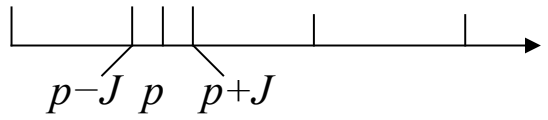
Real-time calculus (RTC)/ Modular performance analysis (MPA)

Streams of events important: Examples

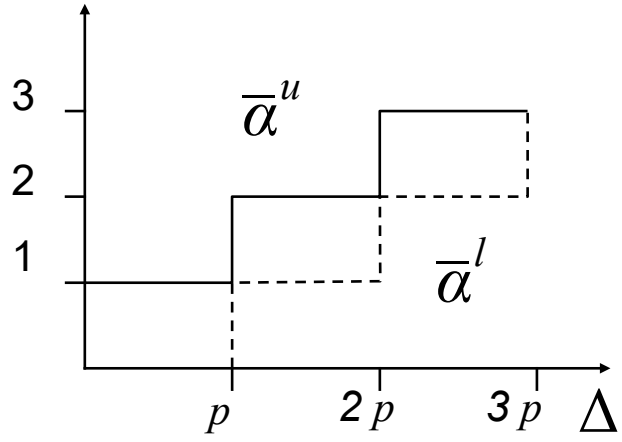
periodic event stream

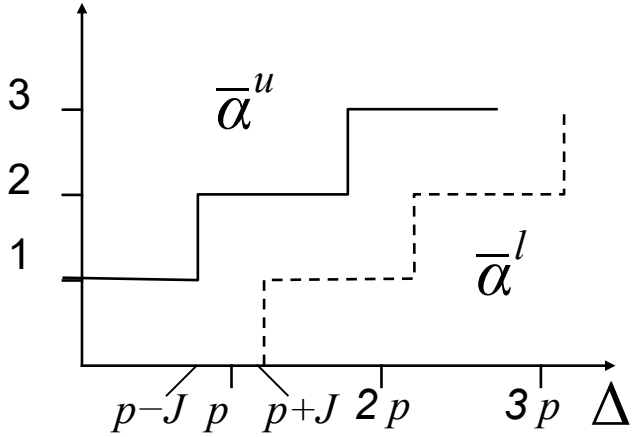


periodic event stream with jitter



Thiele et al. (ETHZ): Extended **network calculus:**
Arrival curves describe the maximum and minimum number of events arriving in some time interval Δ.

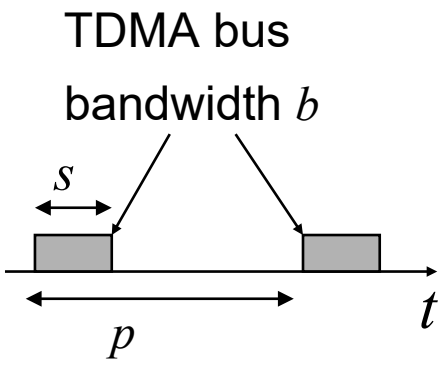


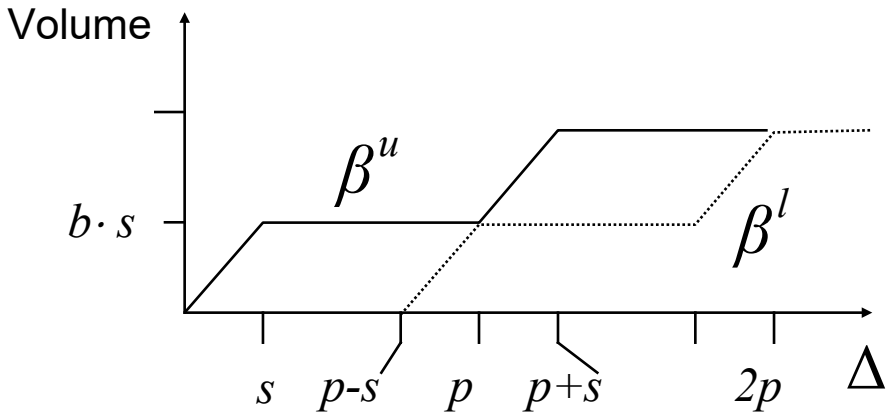


RTC/MPA: Service curves

Service curves β^u resp. β^l describe the maximum and minimum service capacity available in some time interval Δ

Example:

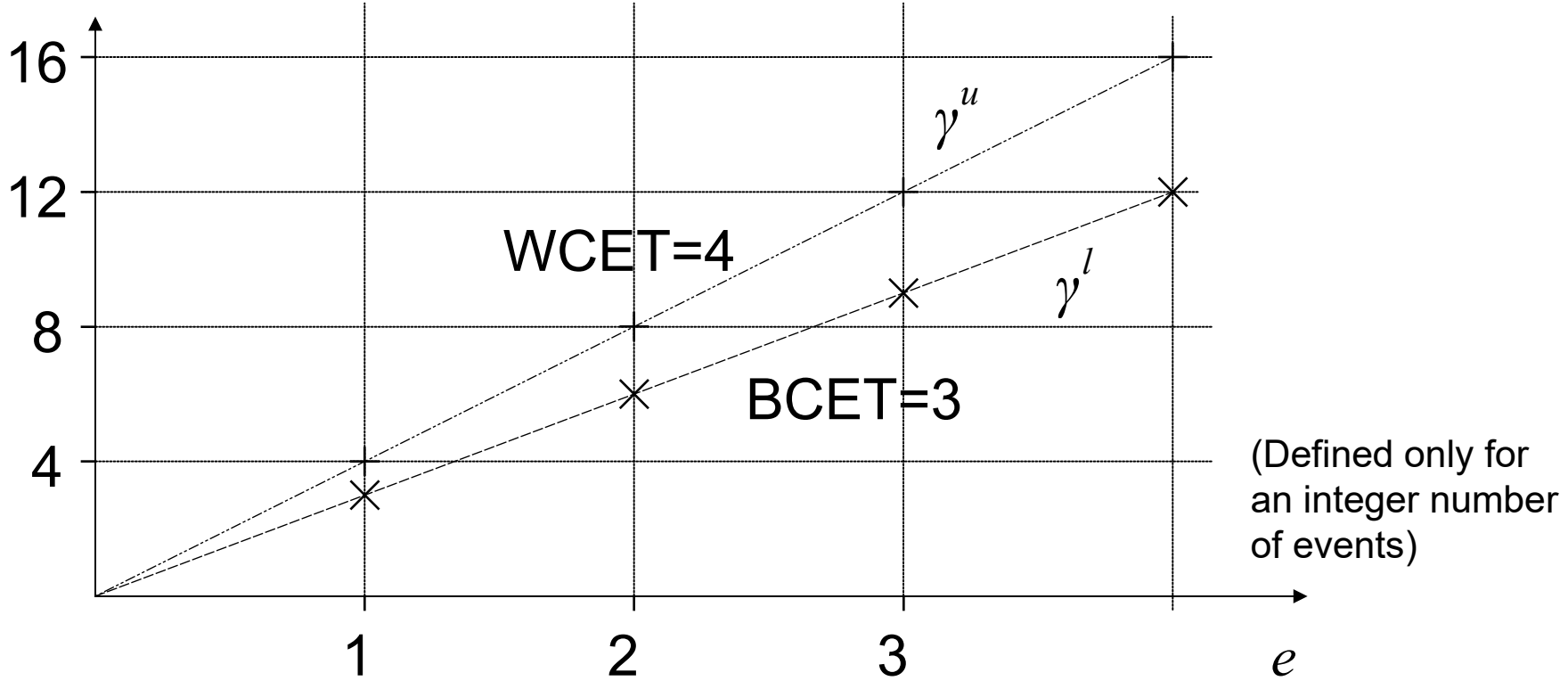




RTC/MPA: Workload characterization

γ^u resp. γ^l describe the maximum and minimum service capacity required as a function of the number e of events.

Example:



RTC/MPA: Workload required for incoming stream

Incoming workload

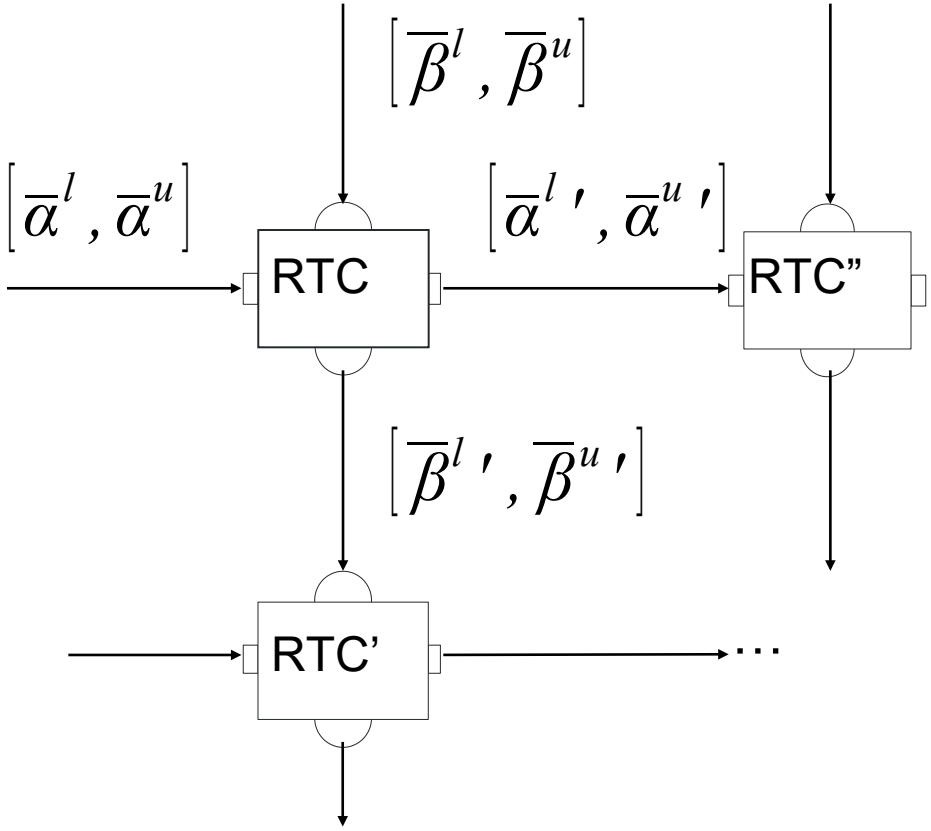
$$\alpha^u(\Delta) = \gamma^u\left(\overline{\alpha}^u(\Delta)\right) \qquad \alpha^l(\Delta) = \gamma^l\left(\overline{\alpha}^l(\Delta)\right)$$

Upper and lower bounds on the number of events

$$\overline{\beta}^u(\Delta) = \left(\gamma^l\right)^{-1}\left(\beta^u(\Delta)\right) \qquad \overline{\beta}^l(\Delta) = \left(\gamma^u\right)^{-1}\left(\beta^l(\Delta)\right)$$

RTC/MPA: System of real time components

Incoming event streams and available capacity are transformed by real-time components:



Theoretical results allow the computation of properties of outgoing streams ☞

RTC/MPA: Transformation of arrival and service curves

Resulting arrival curves:

$$\overline{\alpha}^{u\prime} = \min\left(\left[\left(\overline{\alpha}^{u} \underline{\otimes} \overline{\beta}^{u}\right) \overline{\oplus} \overline{\beta}^{l}\right], \overline{\beta}^{u}\right)$$

$$\overline{\alpha}^{l\prime} = \min\left(\left[\left(\overline{\alpha}^{l} \overline{\oplus} \overline{\beta}^{u}\right) \underline{\otimes} \overline{\beta}^{l}\right], \overline{\beta}^{l}\right)$$

Remaining service curves:

$$\overline{\beta}^{u\prime} = \left(\overline{\beta}^{u} - \overline{\alpha}^{l}\right) \underline{\oplus} 0$$

$$\overline{\beta}^{l\prime} = \left(\overline{\beta}^{l} - \overline{\alpha}^{u}\right) \overline{\otimes} 0$$

Where:

$$\left(f \underline{\otimes} g\right)(t) = \inf_{0 \le u \le t} \left\{f(t-u) + g(u)\right\} \qquad \left(f \overline{\otimes} g\right)(t) = \sup_{0 \le u \le t} \left\{f(t-u) + g(u)\right\}$$

$$\left(f \underline{\oplus} g\right)(t) = \inf_{u \ge 0} \left\{f(t+u) - g(u)\right\} \qquad \left(f \overline{\oplus} g\right)(t) = \sup_{u \ge 0} \left\{f(t+u) - g(u)\right\}$$

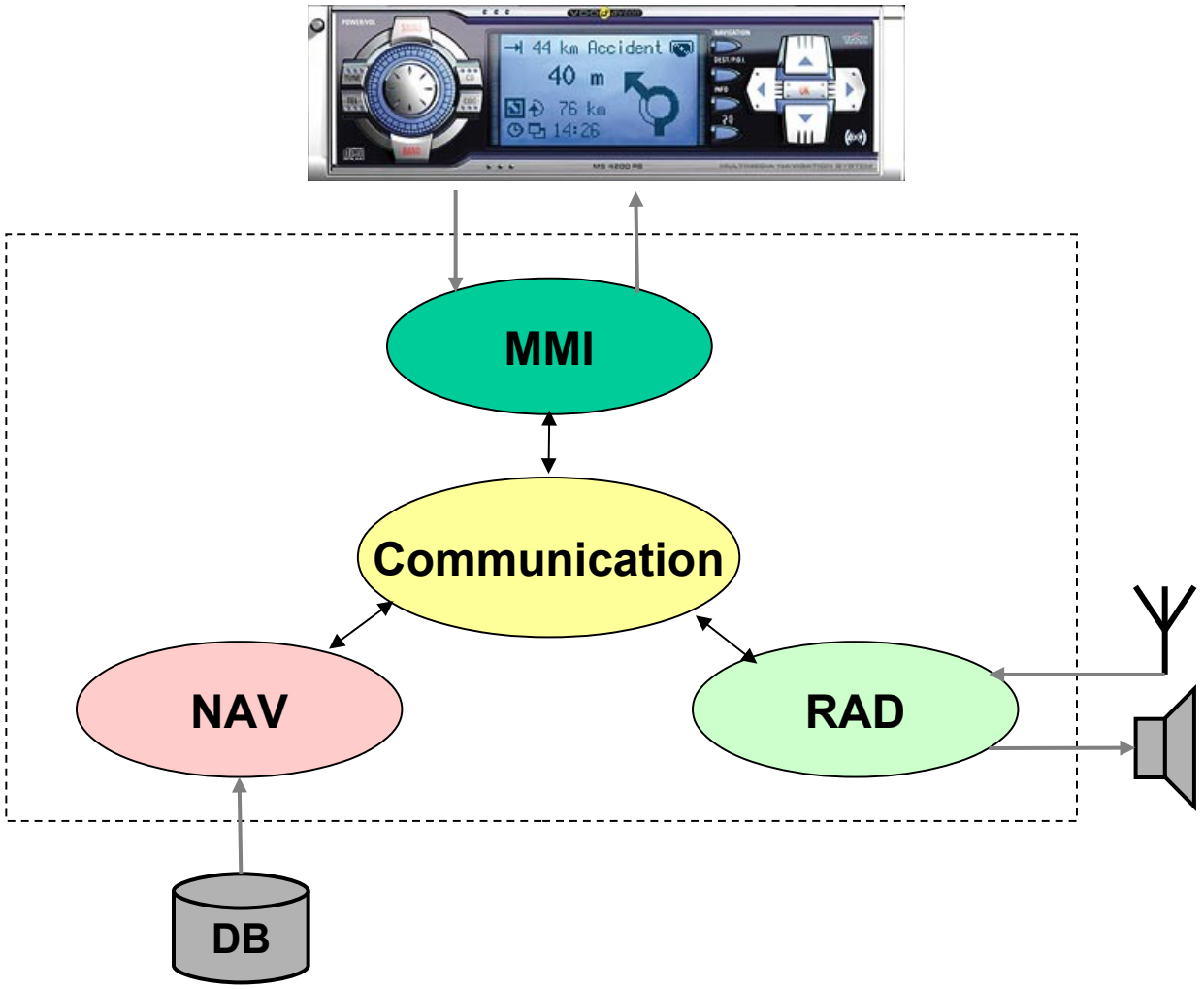
RTC/MPA: Remarks

- Details of the proofs can be found in relevant references
- Results also include bounds on buffer sizes and on maximum latency.
- Theory has been extended into various directions, e.g. for computing remaining battery capacities

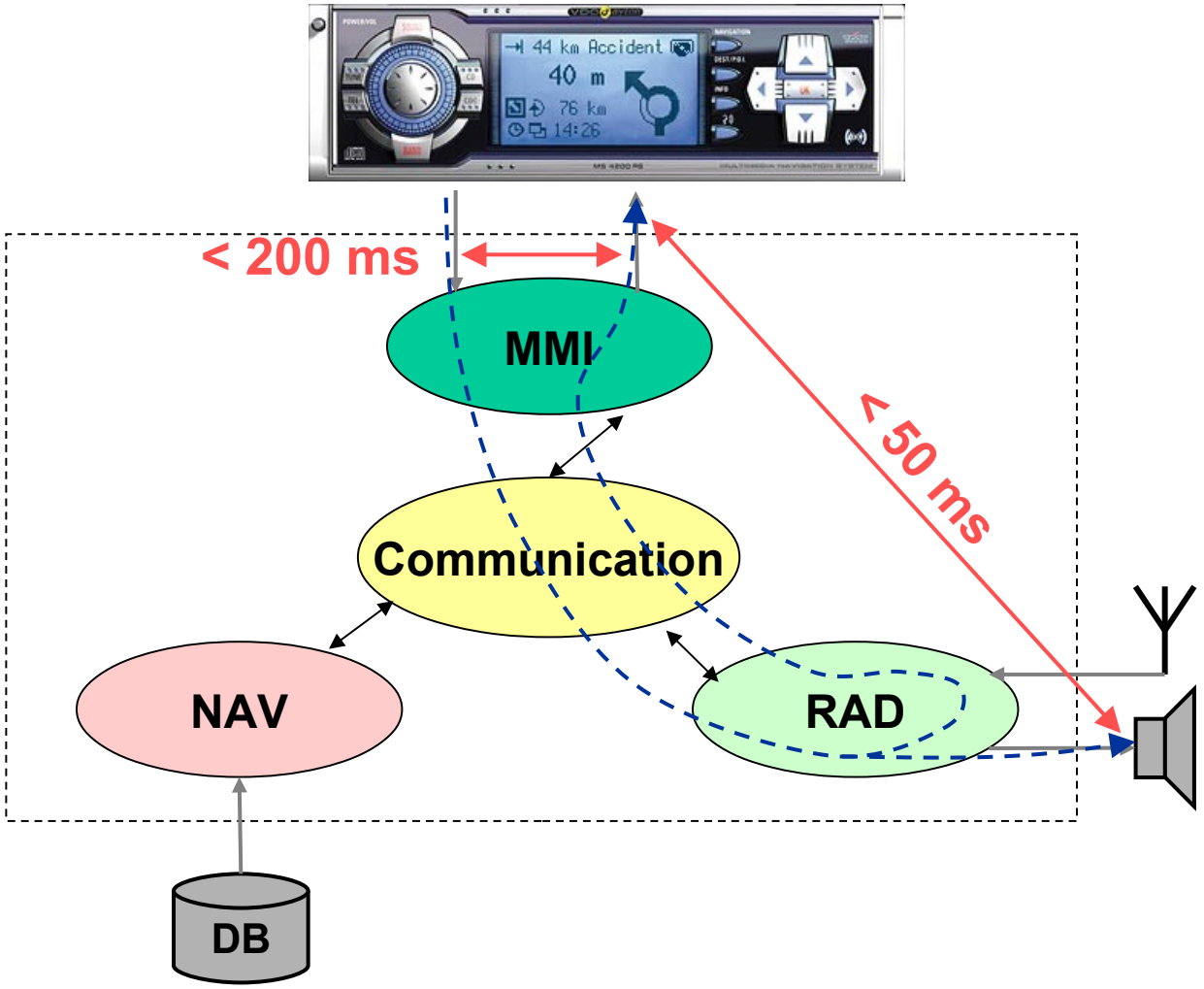
Application: In-Car Navigation System

- Car radio with navigation system
- User interface needs to be responsive
- Traffic messages (TMC) must be processed in a timely way
- Several applications may execute concurrently

System Overview

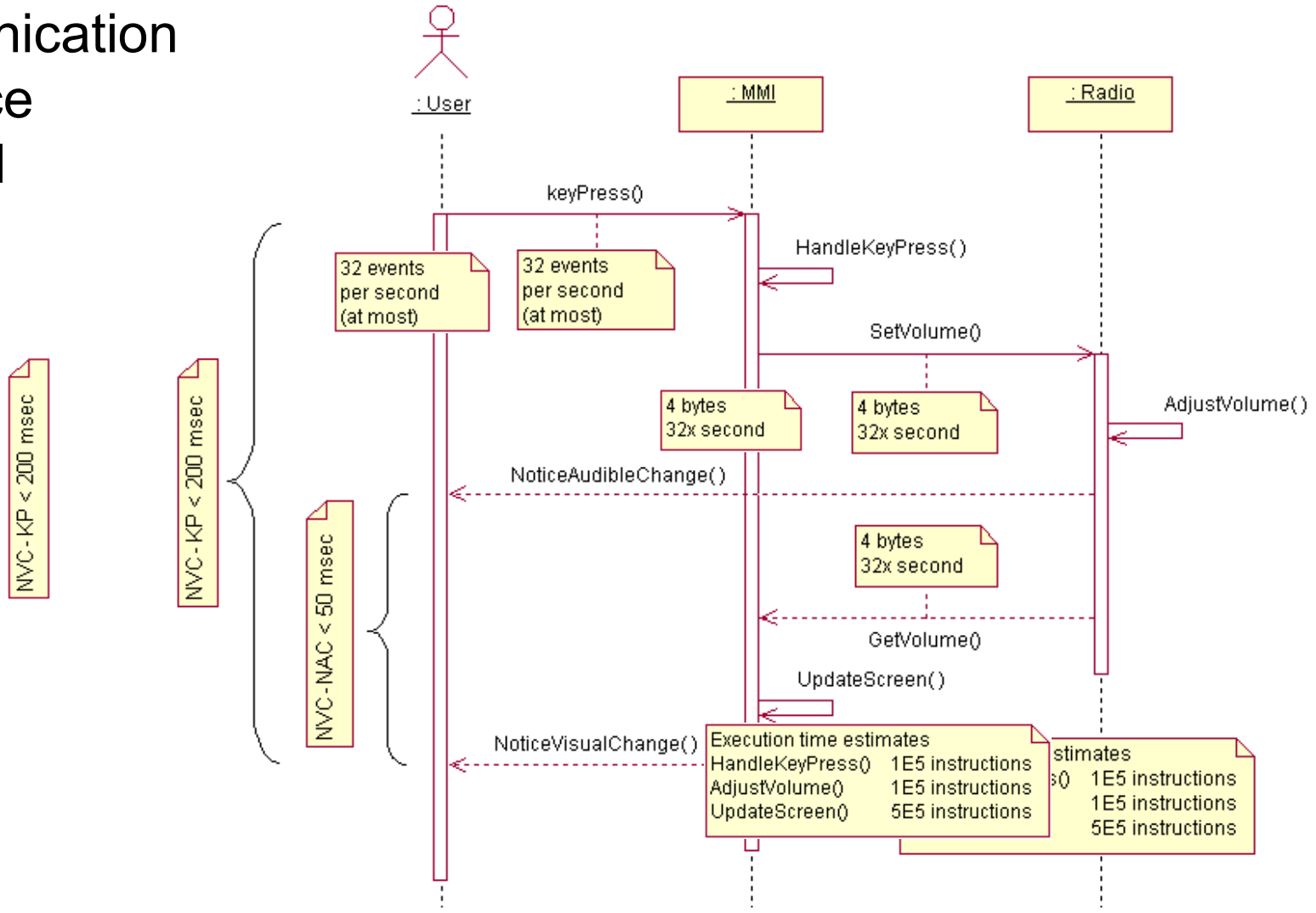


Use case 1: Change Audio Volume (1)

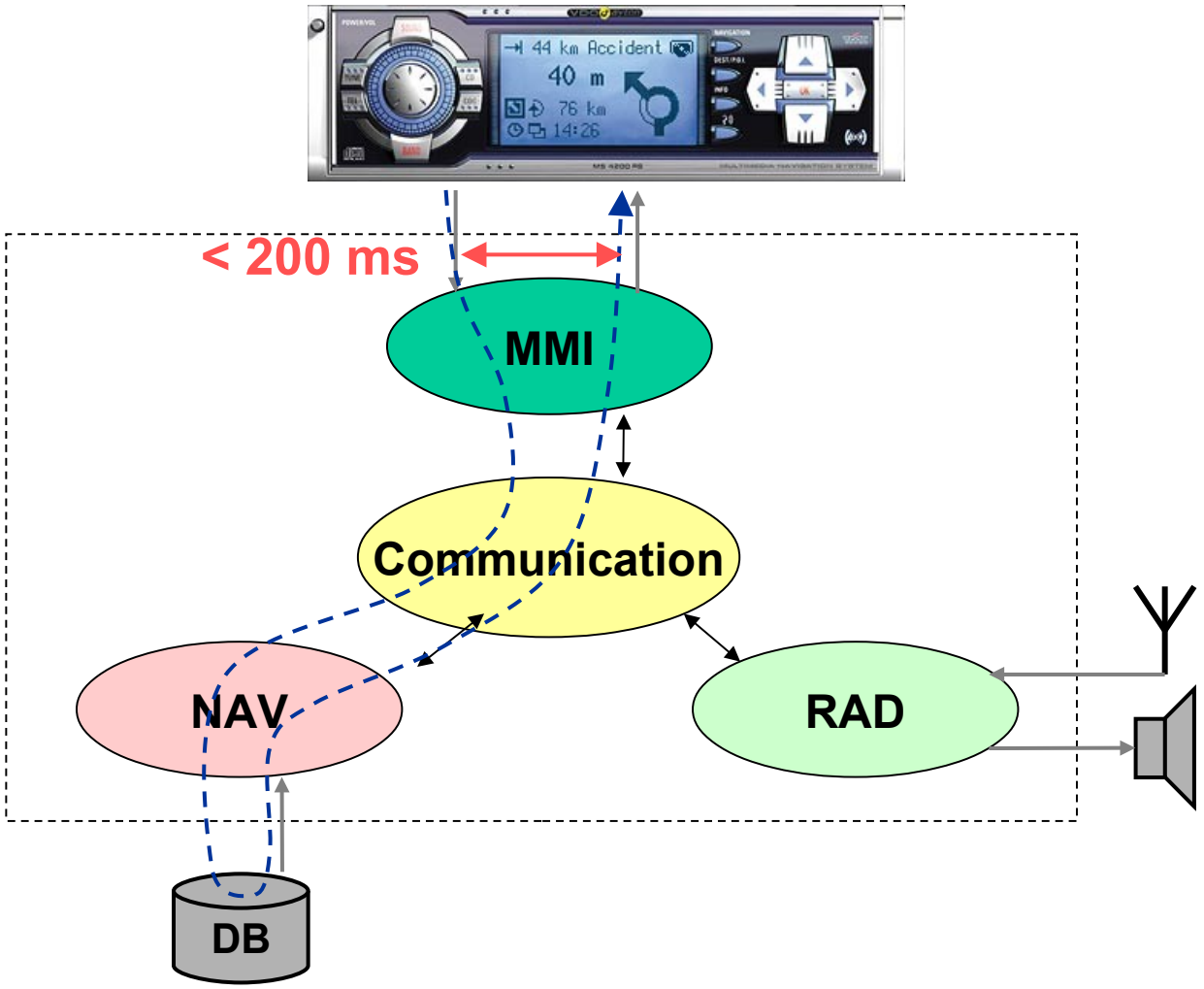


Use case 1: Change Audio Volume (2)

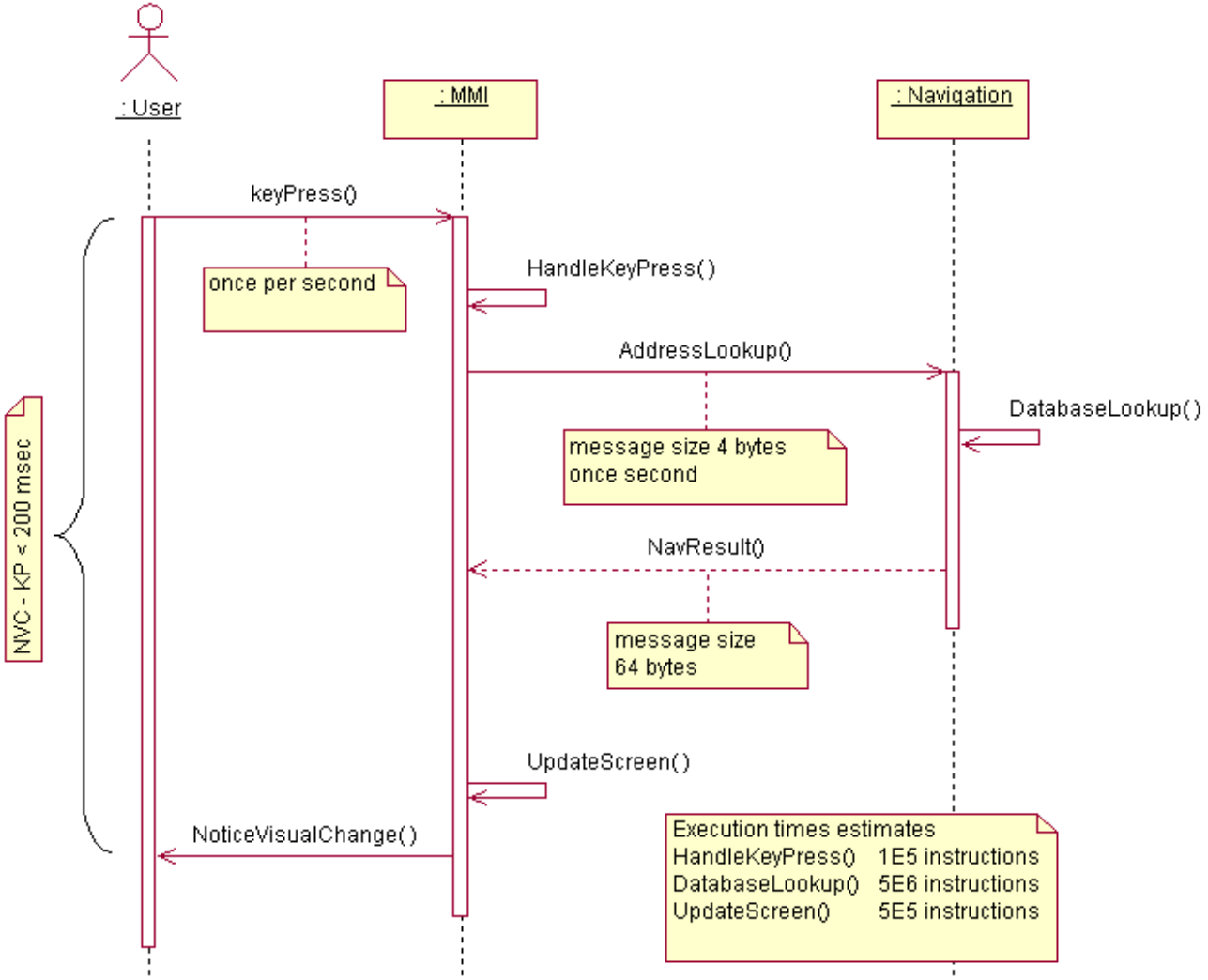
Communication
Resource
Demand



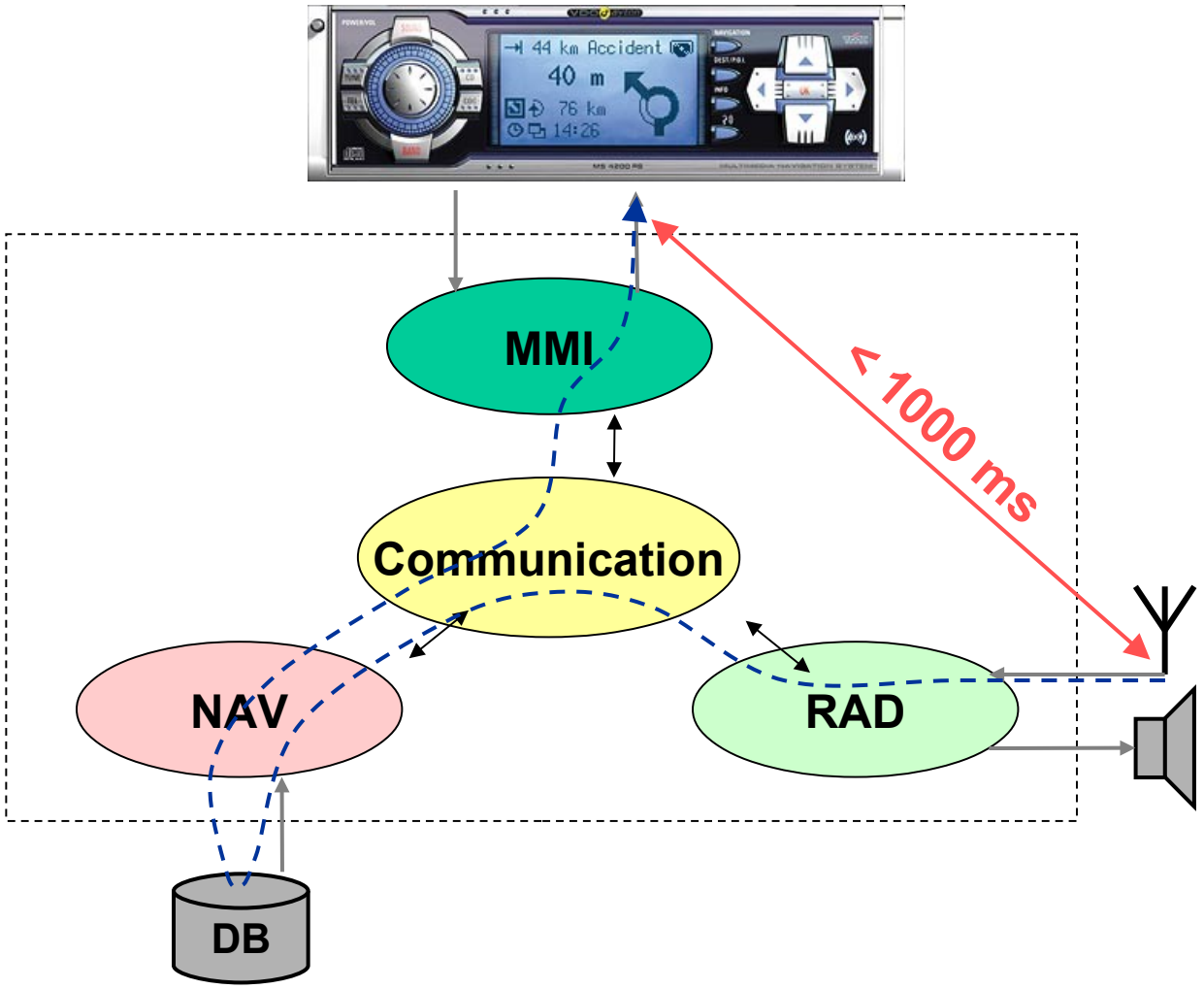
Use case 2: Lookup Destination Address (1)



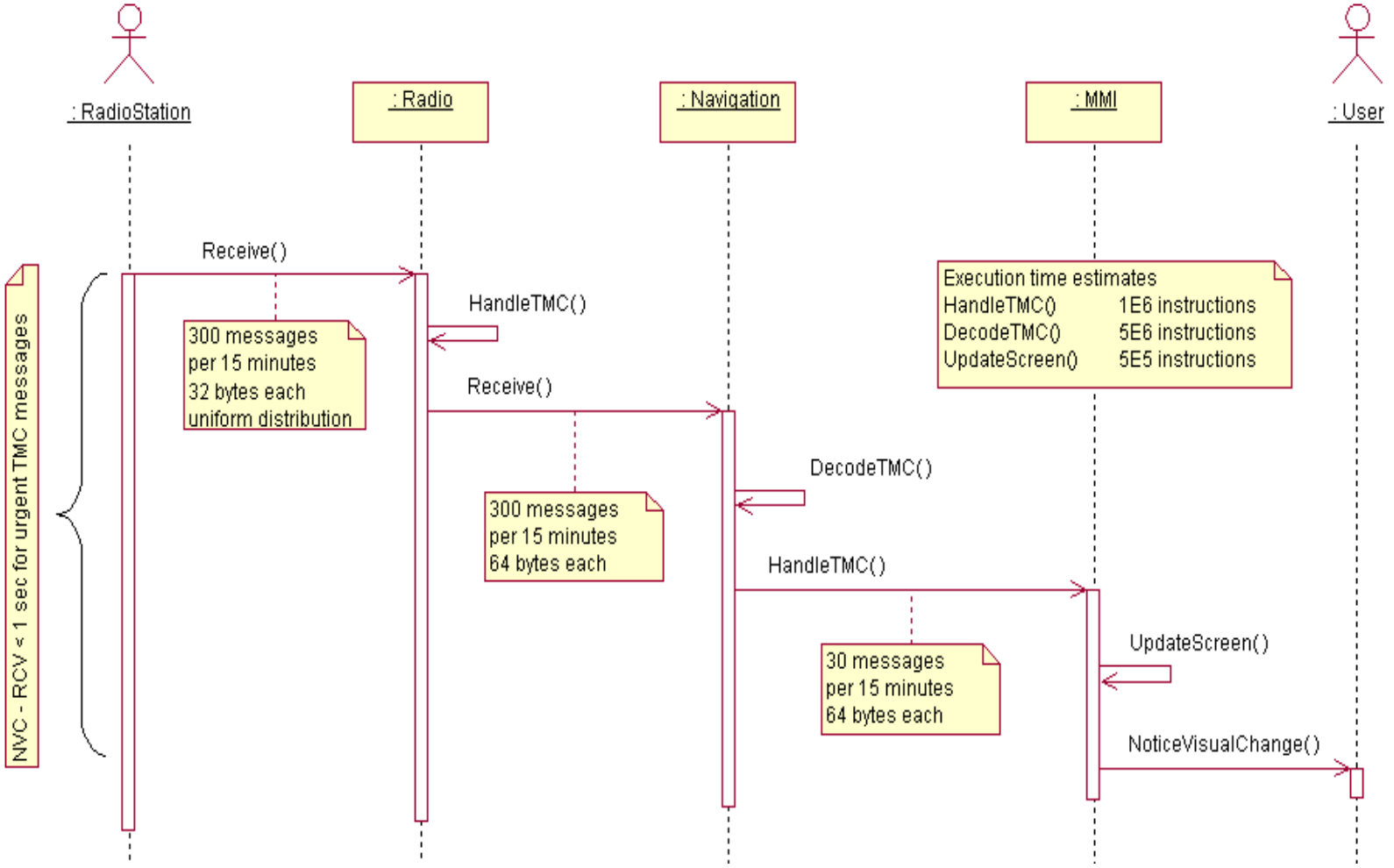
Use case 2: Lookup Destination Address (2)



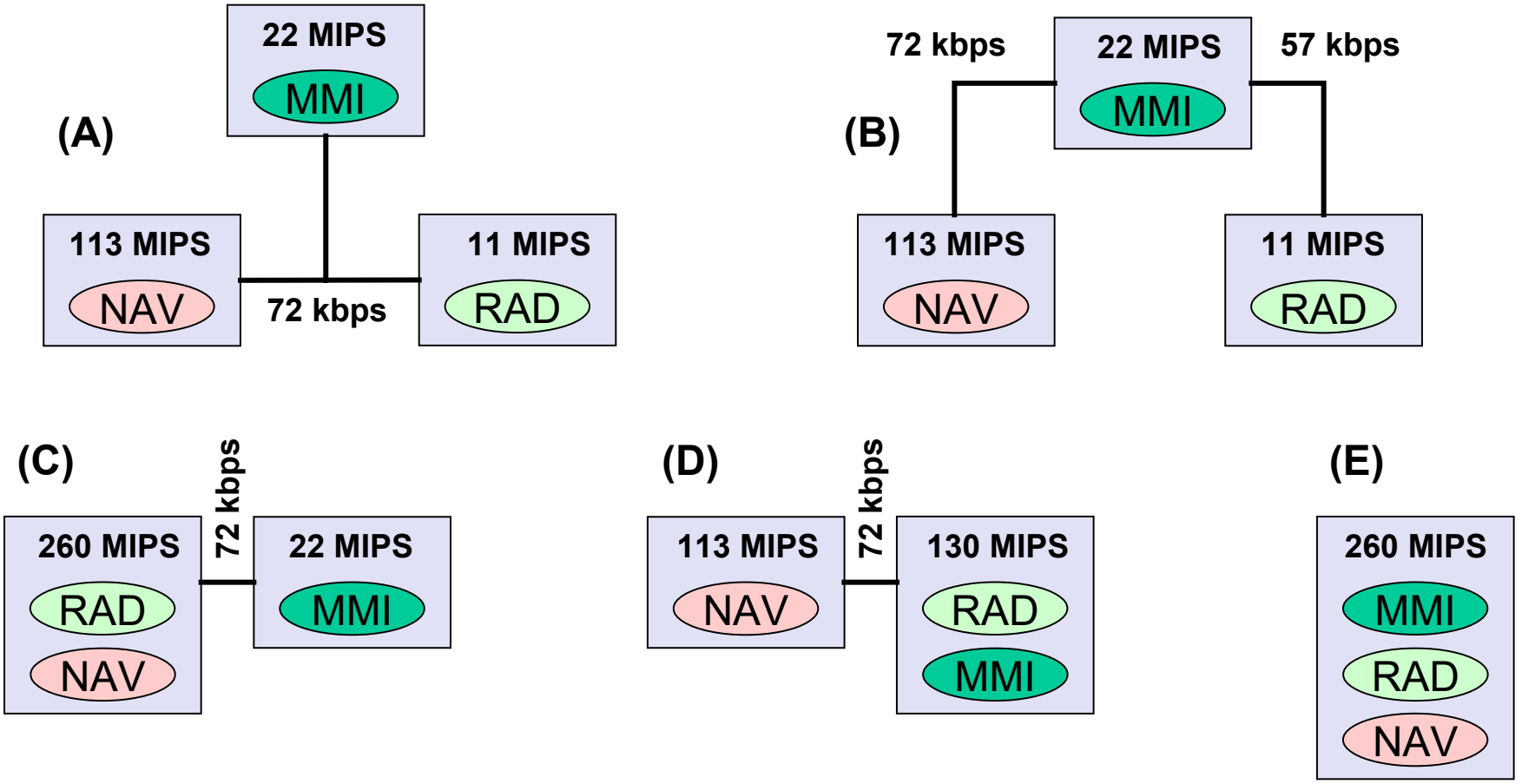
Use case 3: Receive TMC Messages (1)



Use case 3: Receive TMC Messages (2)



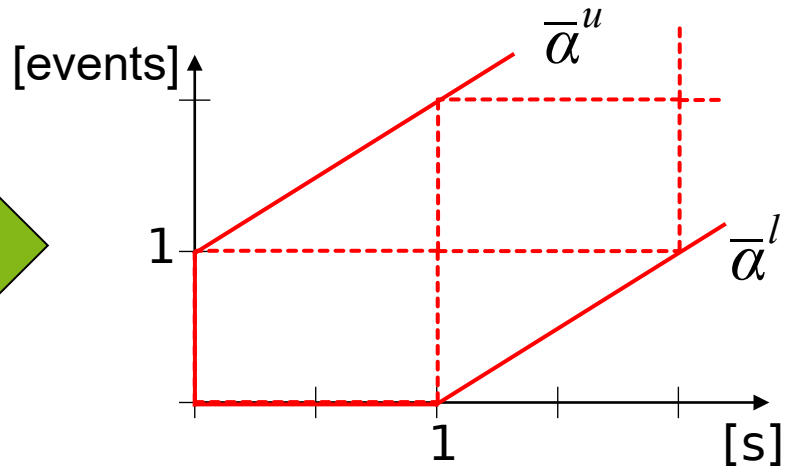
Proposed Architecture Alternatives



Step 1: Environment (Event Steams)

Event Stream Model

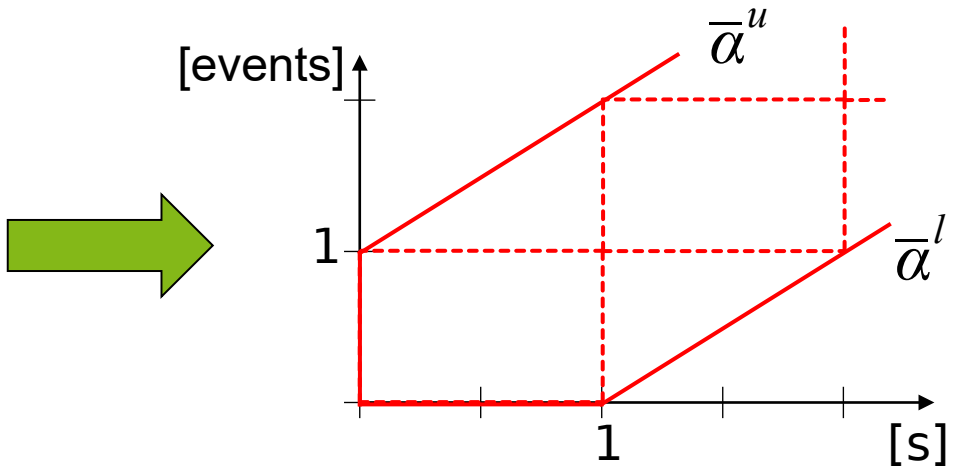
e.g. Address Lookup
(1 event / sec)



Step 2: Architectural Elements

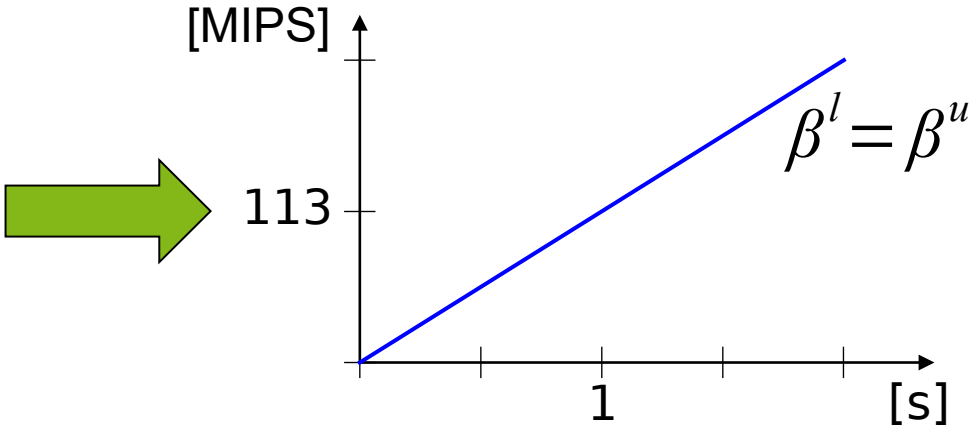
Event Stream Model

e.g. Address Lookup
(1 event / sec)



Resource Model

e.g. unloaded RISC CPU
(113 MIPS)

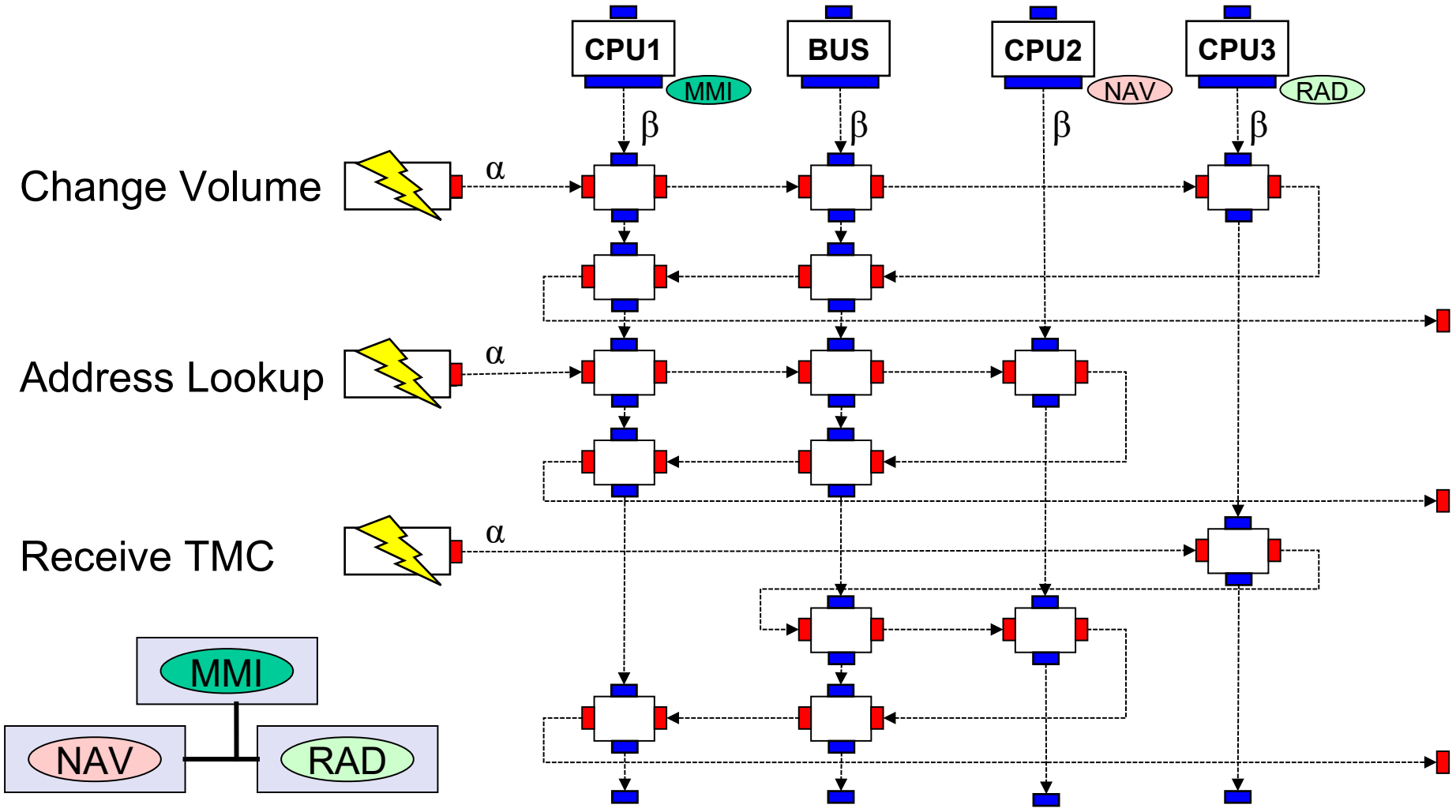


Step 3: Mapping / Scheduling

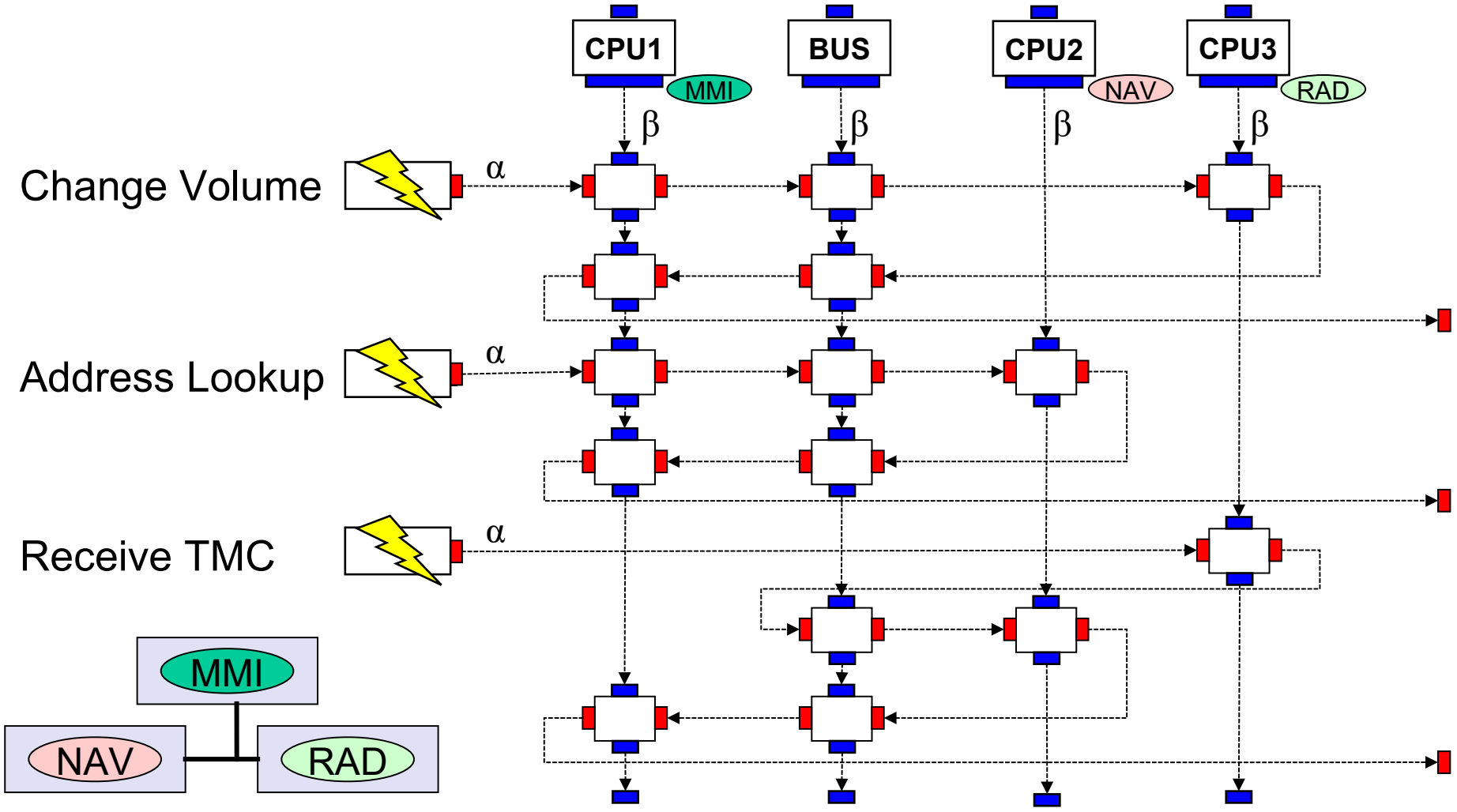
Rate Monotonic Scheduling
(Pre-emptive fixed priority scheduling):

- Priority 1: Change Volume (p=1/32 s)
- Priority 2: Address Lookup (p=1 s)
- Priority 3: Receive TMC (p=6 s)

Step 4: Performance Model



Step 5: Analysis

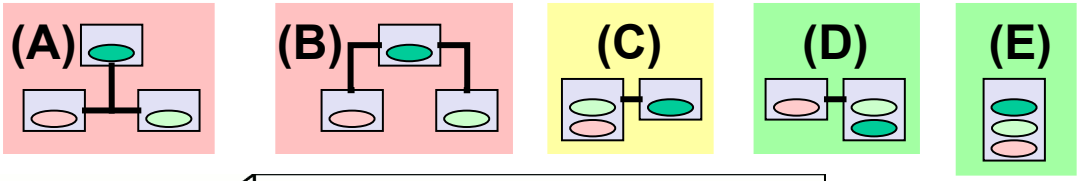


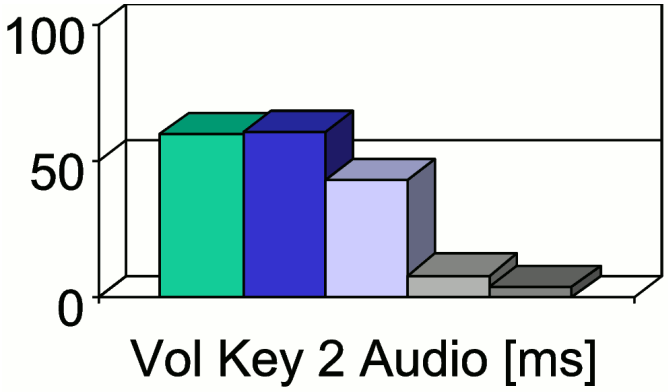
Analysis – Design Question 1

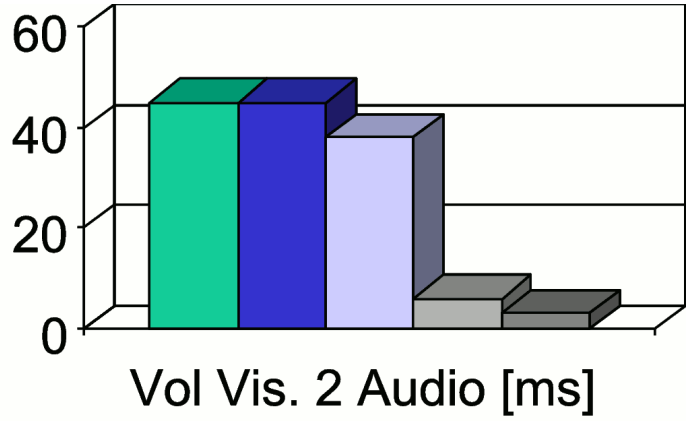
- How do the proposed system architectures compare in respect to end-to-end delays?

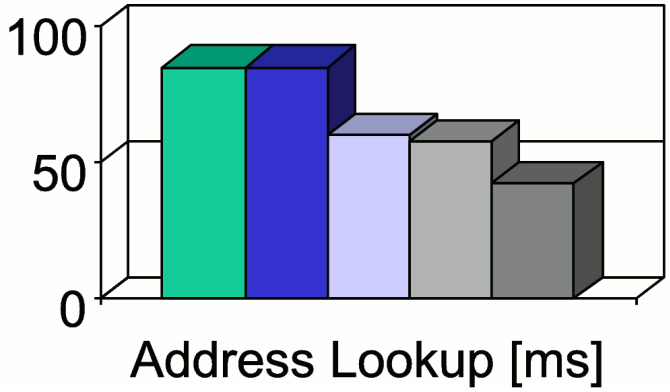
Analysis – Design Question 1

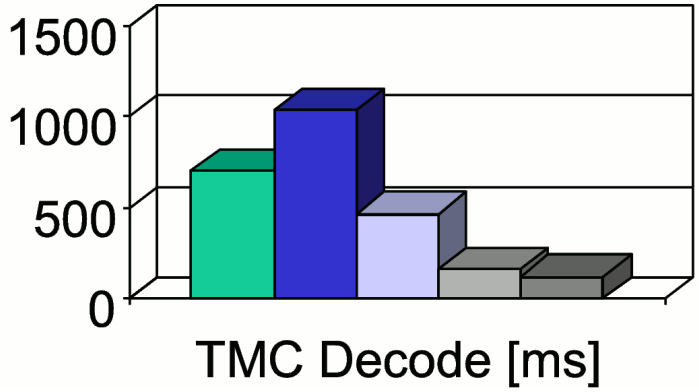
End-to-end delays:





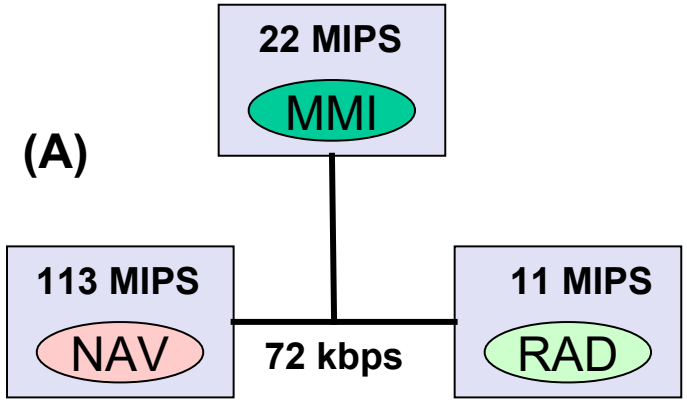






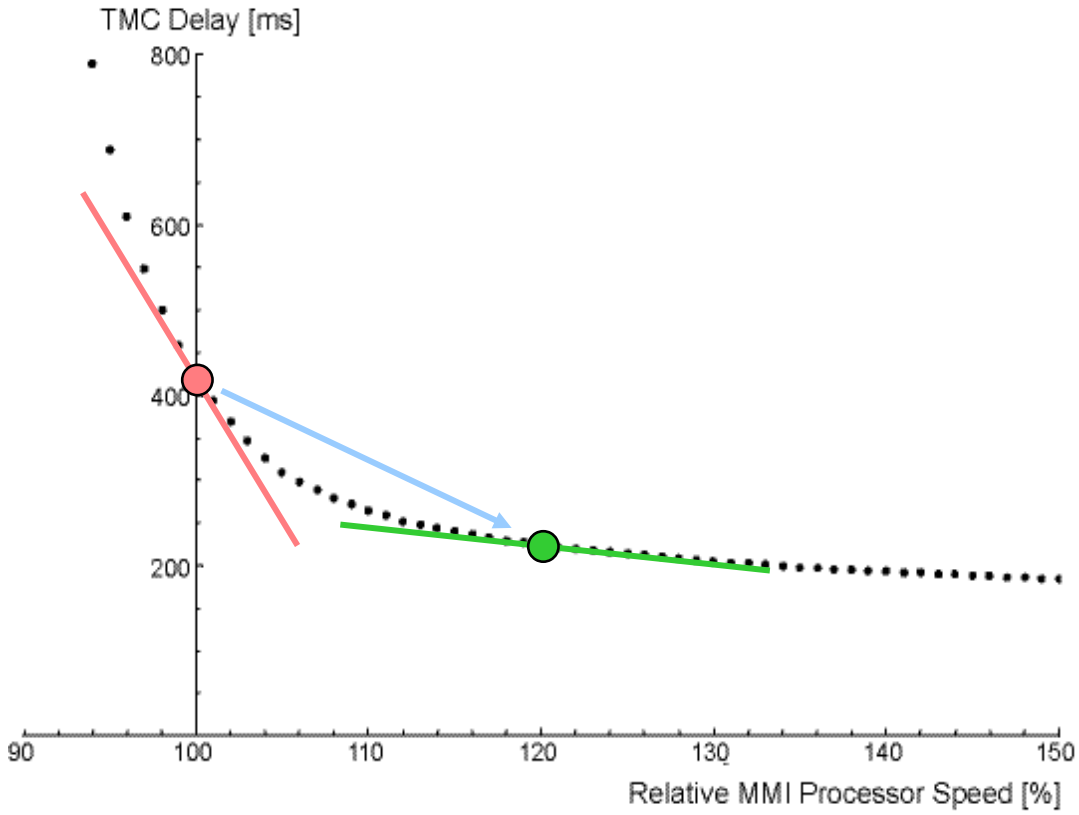
Analysis – Design Question 2

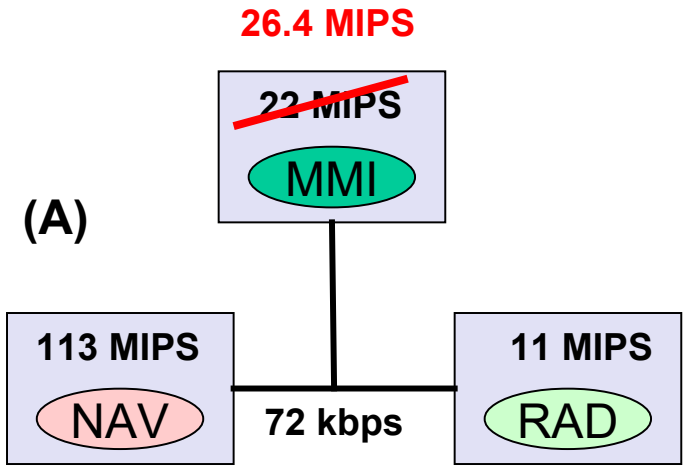
- How robust is architecture A?
- Where is the bottleneck of this architecture?



Analysis – Design Question 2

TMC delay vs. MMI processor speed:





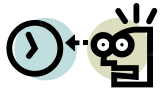
Conclusions

- Easy to construct models (~ half day)
- Evaluation speed is fast and linear to model complexity (~ 1s per evaluation)
- Needs little information to construct early models (Fits early design cycle very well)
- Even though involved mathematics is very complex, the method is easy to use (Language of engineers)

How to evaluate designs according to multiple criteria?

Many different criteria are relevant for evaluating designs:

- Average & worst case delay
- power/energy consumption
- thermal behavior
- reliability, safety, security
- cost, size
- weight
- EMC characteristics
- radiation hardness, environmental friendliness, ..









How to compare different designs?
(Some designs are “better” than others)

Average vs. worst case energy consumption

- The **average energy consumption** E_{AV} is based on the consumption for selected sets of input data (which?)
- The **worst case energy consumption** E_{WC} is a safe upper bound on the energy consumption
- The worst case usage pattern for the battery is ≠ from the worst case for the overall energy consumption
- In general, the pattern for worst case energy consumption is ≠ from the worst case thermal pattern

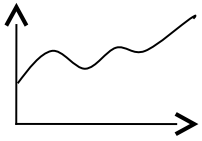
Evaluation of energy consumption: Challenges

- Energy consumption hardly predictable from the source code, due to difficult to predict impact of compiler & linker
- Small variations of the code can lead to large variations of the energy consumption
  - ex. notorious examples
  - Example: shifting code in memory by one byte

- Energy consumption must be predicted from executable code (like the $WCET$)
- The energy consumption might even depend very much on which instance of the hardware is used

Energy models

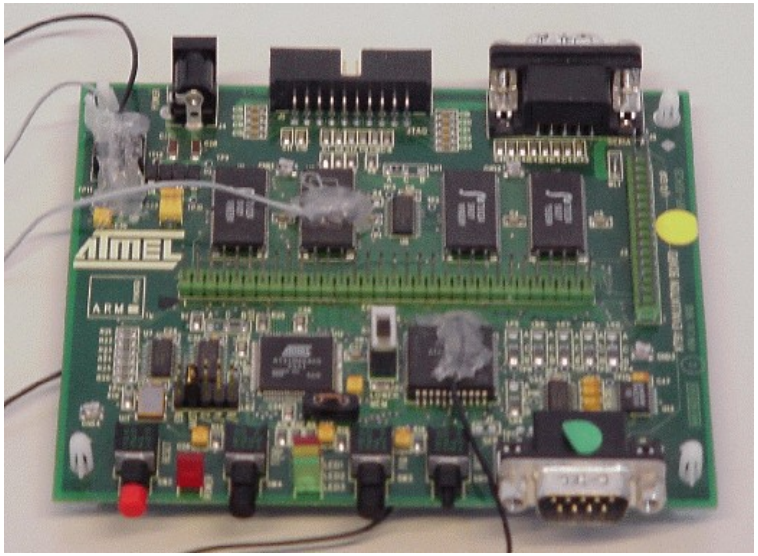
- Measurements: (potentially) precise, fixed architecture
- Models: flexible architecture, less precise
- Combined models



In general, accuracy remains a problem

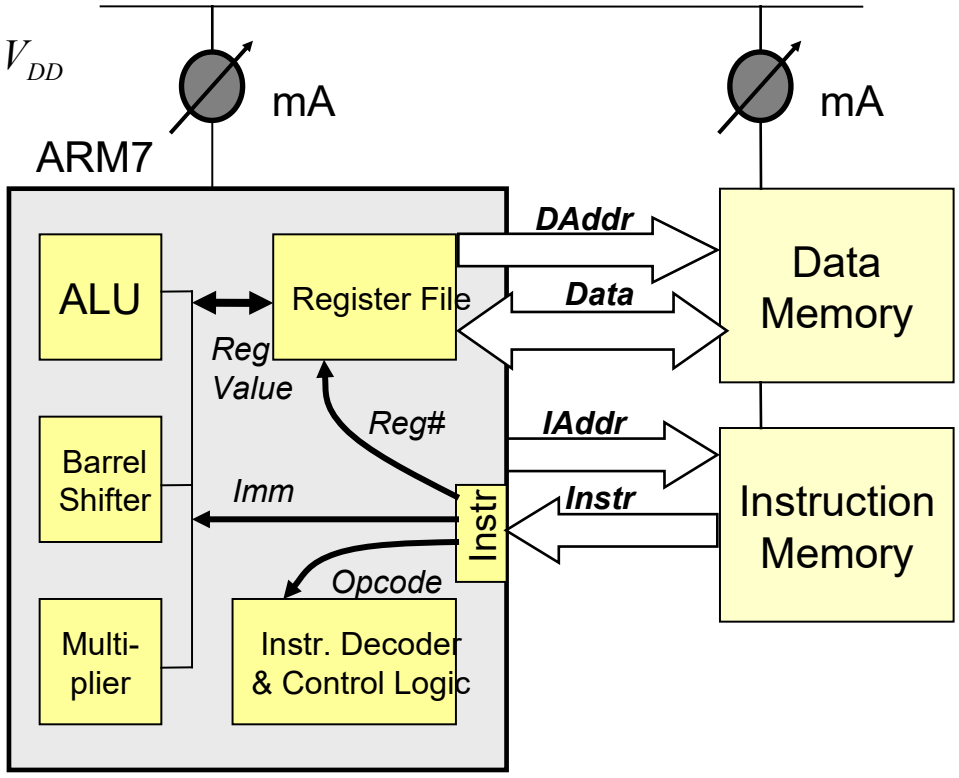
- Currents difficult to measure

Steinke's model



$$E_{total} = E_{cpu\_instr} + E_{cpu\_data} + E_{mem\_instr} + E_{mem\_data}$$

E.g.: ATMEL board with ARM7TDMI and ext. SRAM



S. Steinke, M. Knauer, L. Wehmeyer, P. Marwedel: An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations, Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), 2001

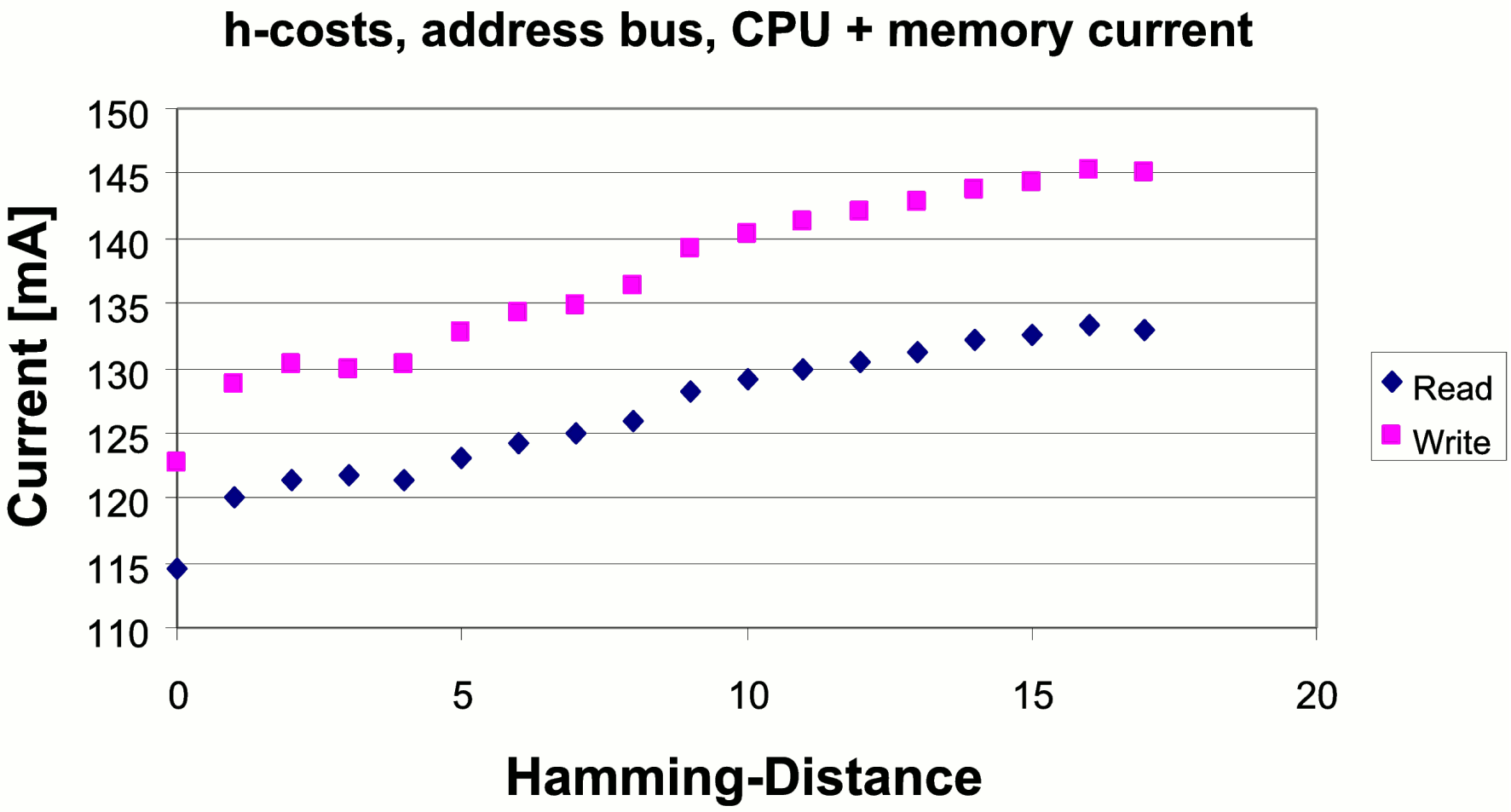
Example: Instruction dependent costs in the CPU

Cost for a sequence of m instructions

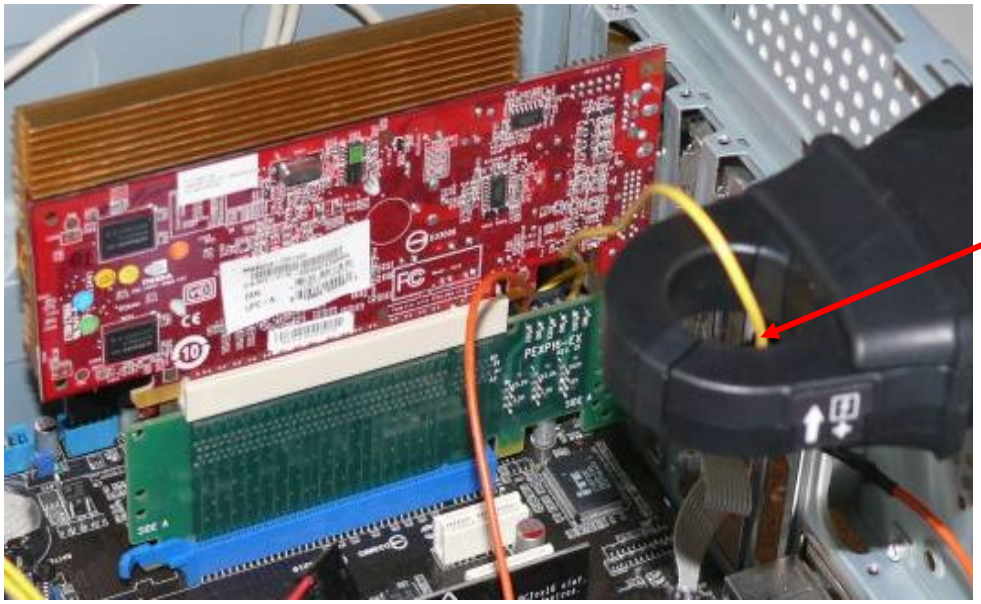
$$E_{cpu\_instr} = \Sigma\, MinCostCPU(\mathbf{Opcode}_i) + FUCost(\mathbf{Instr}_{i-1}, \mathbf{Instr}_i) +$$
$$\alpha_1 * \Sigma\, w(\mathbf{Imm}_{i,j}) + \beta_1 * \Sigma\, h(\mathbf{Imm}_{i-1,j}, \mathbf{Imm}_{i,j}) +$$
$$\alpha_2 * \Sigma\, w(\mathbf{Reg}_{i,k}) + \beta_2 * \Sigma\, h(\mathbf{Reg}_{i-1,k}, \mathbf{Reg}_{i,k}) +$$
$$\alpha_3 * \Sigma\, w(\mathbf{RegVal}_{i,k}) + \beta_3 * \Sigma\, h(\mathbf{RegVal}_{i-1,k}, \mathbf{RegVal}_{i,k}) +$$
$$\alpha_4 * \Sigma\, w(\mathbf{IAddr}_i) + \beta_4 * \Sigma\, h(\mathbf{IAddr}_{i-1}, \mathbf{IAddr}_i)$$

w:	number of ones;
h:	Hamming distance;
$FUCost$:	cost of switching functional units;
α, β:	determined through experiments.

Hamming Distance between adjacent addresses is playing major role



Energy-efficient execution on graphics processor (GPU)



current clamp

Energy per frame CPU	3.26 J	5.84 J	10.52 J	Reduced to
Energy per frame GPU	0.93 J	1.56 J	2.76 J	avg 27%

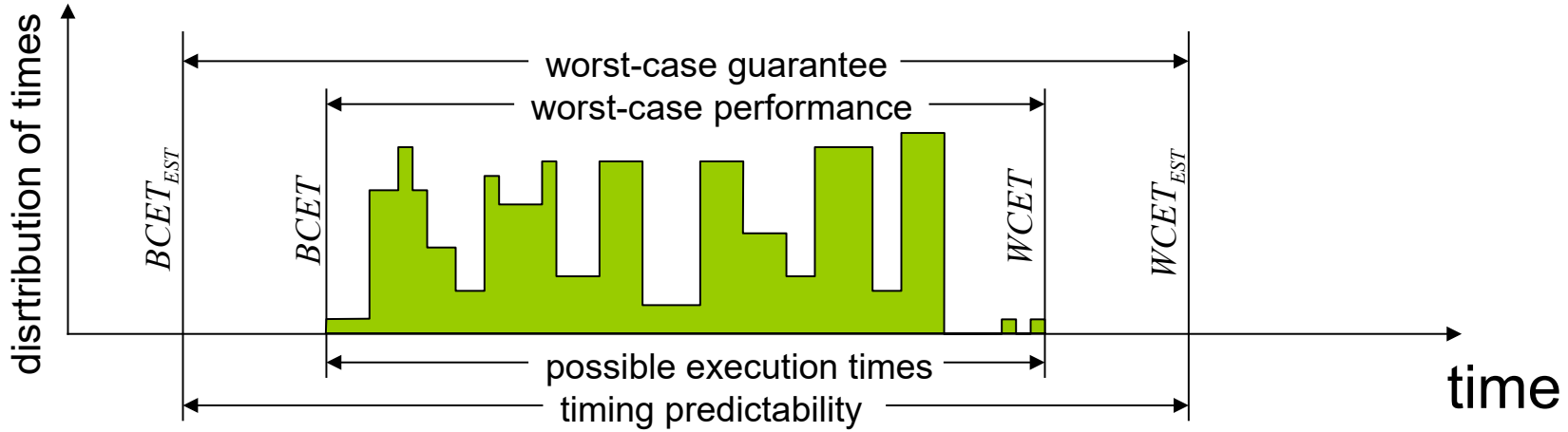
C. Timm, A. Gelenberg, P. Marwedel, F. Weichert: Energy Considerations within the Integration of General Purpose GPUs in Embedded Systems. Intern. Conf. on Advances in Distributed and Parallel Computing, 2010

C. Timm, F. Weichert, P. Marwedel, H. Müller: Design Space Exploration Towards a Realtime and Energy-Aware GPGPU-based Analysis of Biosensor Data. Computer Science - Research and Development, ENA-HPC, 2011

Examples of energy models

- Measurements:
  - Tiwari (1994): Energy consumption within processors
  - Russell, Jacome (1998): Measurements for 2 fixed configurations
  - Simunic (1999): Values from data sheets. Not very precise.
  - Timm: measurements for graphics card
- Models:
  - CACTI [Jouppi, 1996]: Predicted energy consumption of caches
  - Wattch [Brooks, 2000]: Power estimation at the architectural level, without circuit or layout, known to be imprecise
- Combined models
  - Steinke et al., TU Dortmund (2001): mixed model

Worst case energy consumption via worst case computing time?



- Computing the E_{WC} using $WCET$ estimations

$$E_{WC} = \int_0^{WCET_{EST}} P(t)'\,dt$$

- Tight bounds if $P(t)$ has small variations & $WCET_{EST}$ is tight
- Little value if $P(t)$ varies too much.

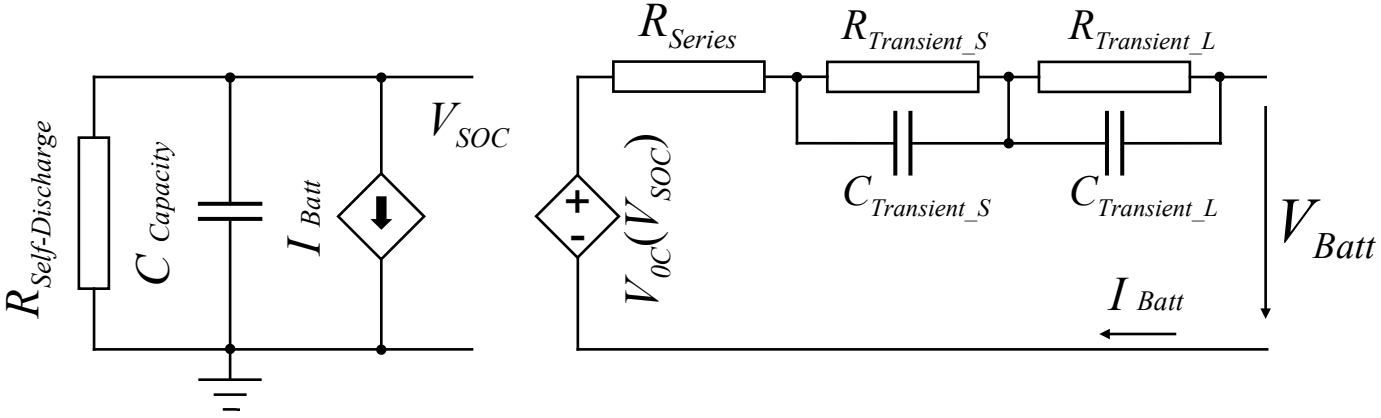
Battery models

- (Chemical) & physical models
  e.g. concentrated solution theory, partial differential eq.s
  many, frequently unknown parameters (50+); xy hours simulation time
- Empirical models
  Simple equations, inaccurate
  - Peukert's law: lifetime= C/I^{α}, with empirical α
  - Weibull fit
- Abstract models
  - Electrical circuit models
  - Discrete time model (e.g. in VHDL)
  - Stochastic models (e.g. Markov processes)
- Mixed models
  e.g. electrical models with physical explanation

frequently with fitting

Sources include: R. Rao, S. Vrudhula, D.N. Rakhmatov: Battery Modeling for Energy-Aware System Design, IEEE Computer, 2003, pp. 77

Model proposed by Chen and Rincón-Mora

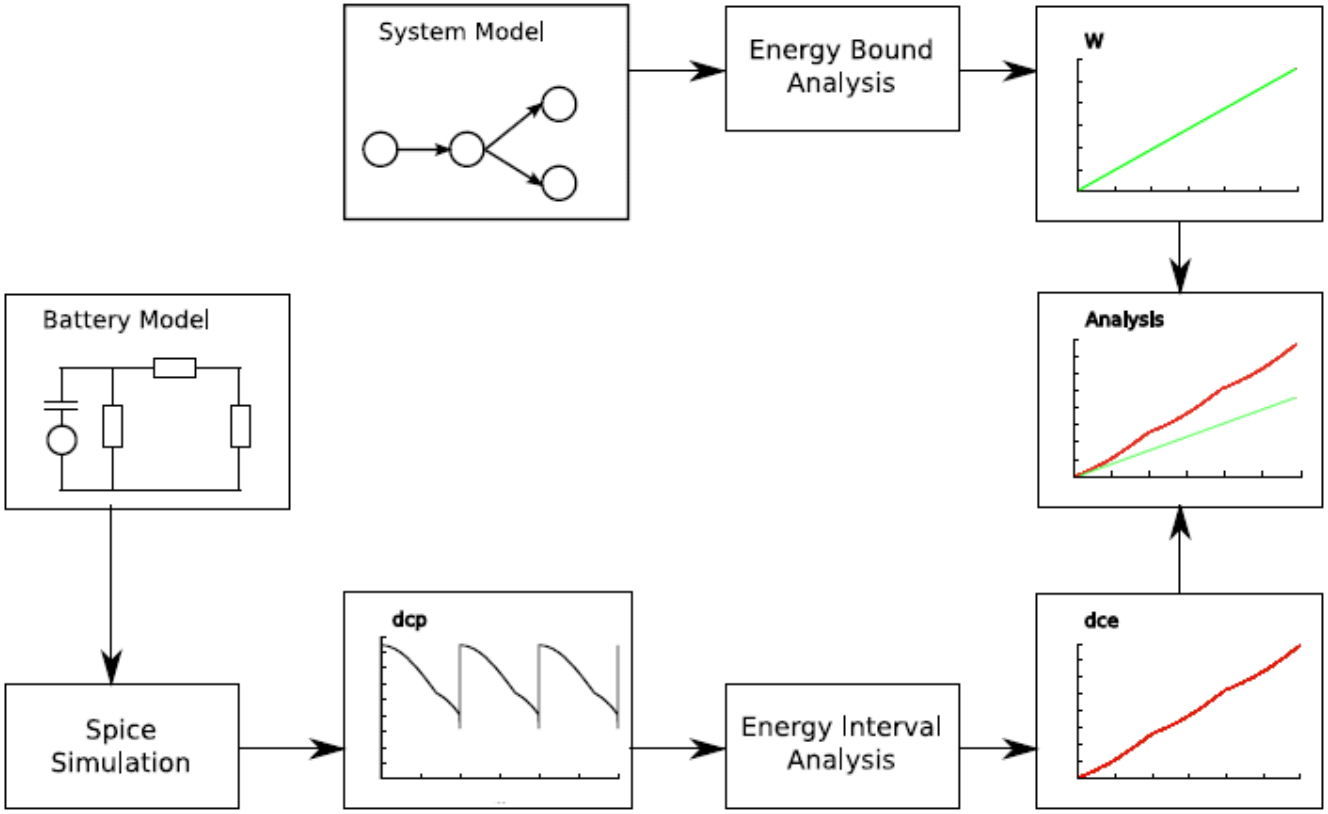


Source: M. Chen, G. A. Rincón-Mora: Accurate Electrical Battery Model Capable of Predicting Runtime and *I-V* Performance, *IEEE Trans. on Energy Conversion*, 2006, pp. 504

- Full charge capacitor: $C_{Capacity} = 3600 * Capacity * f_1(\text{cycle}) * f_2(\text{Temp})$
- Self-discharge resistor: $R_{Self\text{-}Discharge}$ (might depend on parameters)
- Current dependency of V_{Batt}: modeled by $R_{series} + R_{transient\_S} + R_{Transient\_L}$
- I_{Batt} charges and discharges $C_{Capacity}$
- Voltage controlled voltage source V_{0C} captures nonlinear dependency between the state of charge and V_{0C} (measurement can take days)
- R_{Series}: models immediate voltage drop at load change

Battery capacity sufficient?

Question can be solved with adapted real-time calculus

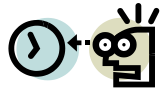


Lipskoch, H., Albers, K. and Slomka, F.: Battery discharge aware energy feasibility analysis, Proceedings of the 4th international Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '06, pp. 22-27, 2006.

How to evaluate designs according to multiple criteria?

Many different criteria are relevant for evaluating designs:

- Average & worst case delay
- power/energy consumption
- thermal behavior
- reliability, safety, security
- cost, size
- weight
- EMC characteristics
- radiation hardness, environmental friendliness, ..









How to compare different designs?
(Some designs are “better” than others)

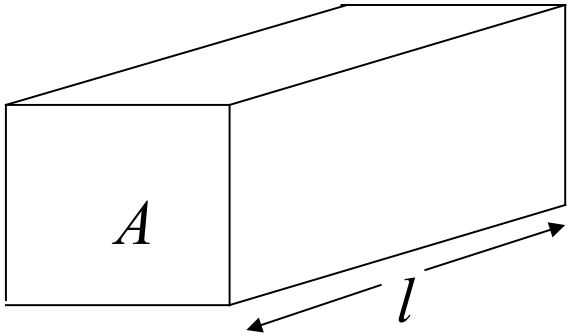
Thermal models

Thermal models becoming increasingly important

- since temperatures become more relevant due to increased performance, and
- since temperatures affect
  - usability and
  - dependability.

Thermal conductivity

$$P_{th} = \kappa \frac{\Delta T \cdot A}{l} \qquad (1)$$



Where

P_{th}	: thermal power transferred
κ	: thermal conductivity
ΔT	: temperature difference
A	: area
l	: length

Thermal conductivity κ reflects the amount of thermal energy per unit of time transferred through a plate made of some material of area A and thickness l when the temperatures at the opposite sides differ by one temperature unit (e.g. Kelvin)

Examples of thermal conductivity

Material	**Thermal conductivity** [W/(m K)]
Copper	240-401
Aluminum (95.5%)	236
Silicon	148
Wood (perpendicular to fibre)	0.09-0.19
Concrete	0.08-0.25
Air (21% oxygen)	0.0262






https://de.wikipedia.org/wiki/Wärmeleitfähigkeit

Thermal conductance & resistance

- **Thermal conductance** = amount of thermal energy which passes through a plate per unit of time if the temperatures at the two ends of the plate differ by one unit of temperature (e.g. Kelvin).

$$P_{th} = \kappa \frac{\Delta T \cdot A}{l} \qquad (1) \quad \rightarrow \quad \frac{P_{th}}{\Delta T} = \kappa \frac{A}{l} \qquad (2)$$

- The reciprocal of thermal conductance is called **thermal resistance** R_{th}.

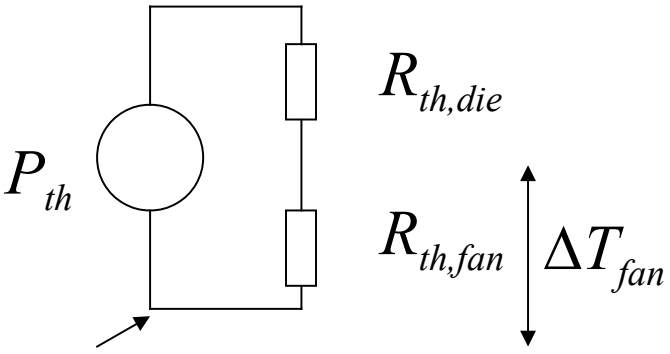
$$R_{th} = \frac{\Delta T}{P_{th}} = \frac{l}{\kappa \cdot A} \qquad (3)$$

Equivalent thermal circuits

- Thermal resistances add up like electrical resistances

☞ Thermal modeling mapped to circuit modeling

e.g.: microprocessor:



Ground ≈ Reference temperature

$$\Delta T = R_{th} \cdot P_{th} \qquad (3)$$

$$R_{th} = R_{th,die} + R_{th,fan} \qquad (4)$$

For $R_{th,die}$ = 0.4 [W/K],
$R_{th,fan}$ = 0.3 [W/K],
P_{th} = 10 [W]:

☞ ΔT = 7 [K],
ΔT_{fan} = 3 [K]

So far, we have just considered the steady state.

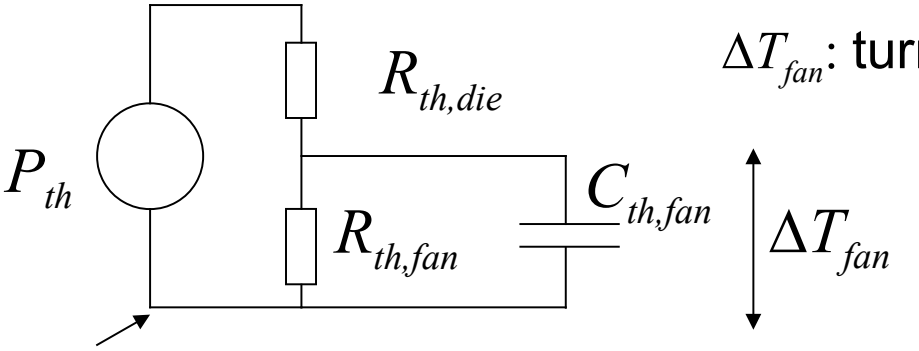
Dynamic thermal properties

In general, transients and thermal capacity to be considered:

$C_{th} = m \cdot c$

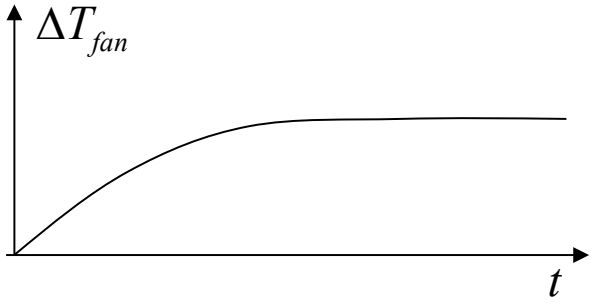
where C_{th}: thermal capacity, m: mass, c: specific heat

☞ Networks comprising resistances and capacities



Ground ≈ Reference temperature

ΔT_{fan}: turned into transient with time constant



Extra voltage source can make reference temperature explicit

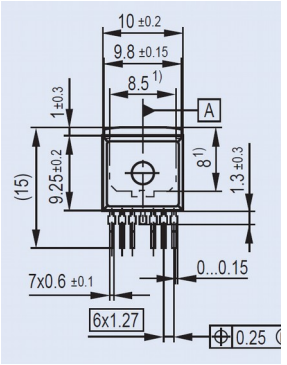
https://www.infineon.com/dgdl/smdpack.PDF?folderId=db3a304412b407950112b417b3e623f4&fileId=db3a304412b407950112b417b42923f5

Equivalences

Electrical model		Thermal model	
Current	I	Thermal flow, flow of “power”	$P_{th} = \dot{Q}$
Total charge	$Q = \int I\, dt$	Thermal energy	$E_{th} = \int P_{th}\, dt$
Resistance	R	Thermal resistance	R_{th}
Potential	φ	Temperature	T
Voltage = potential difference	U	Temperature difference	ΔT
Capacitance	C	Thermal capacitance	C_{th}
Ohms law	$U = R\, I$	Δ Temperature at R_{th}	$\Delta T = R_{th} \cdot P_{th}$

Examples of thermal resistance of P-TO263-7-3

Component	Value & Dimension
Thermal resistance of chip	0.48 [K/W]
Thermal time constant of chip	≈1.5 ms
Thermal capacity of chip	≈3 [mWs/K]
Thermal resistance of heat slug	0.24 [K/W]
Thermal capacity of heat slug	310 [mWs/K]
Thermal time constant of heat slug	70 [ms]



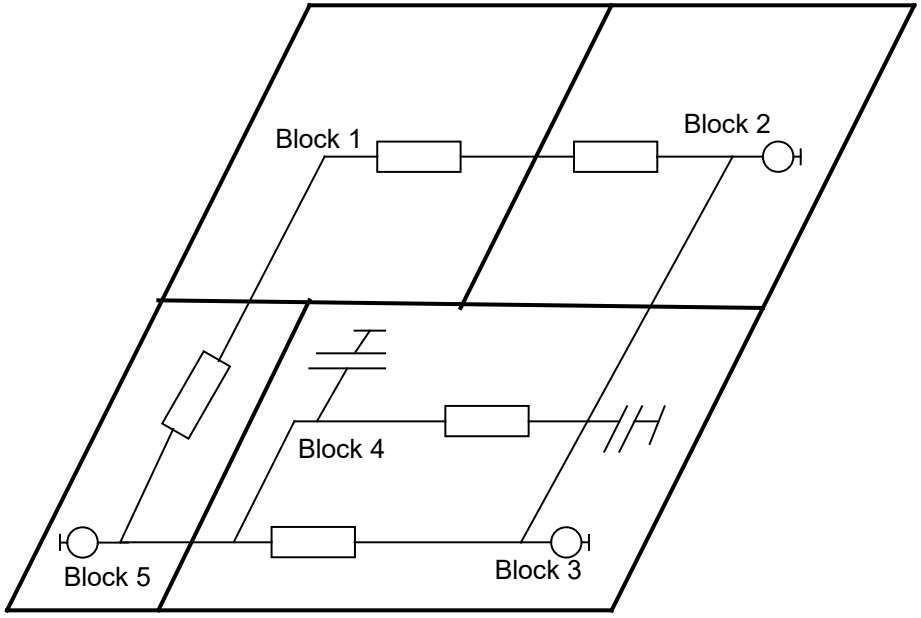
https://www.infineon.com/dgdl/smdpack.PDF?folderId=db3a304412b407950112b417b3e623f4&fileId=db3a304412b407950112b417b42923f5

Hotspot – A popular thermal simulator for processors

- Localized heating much faster than chip-wide (millisec time scale)
- Chip-wide treatment is inaccurate (neglects hot spots)
- Temperature is sensitive to chip layout (floorplan)

☞ Fine-grained, dynamic model of temperature

- Authors say: Validated against FEM models

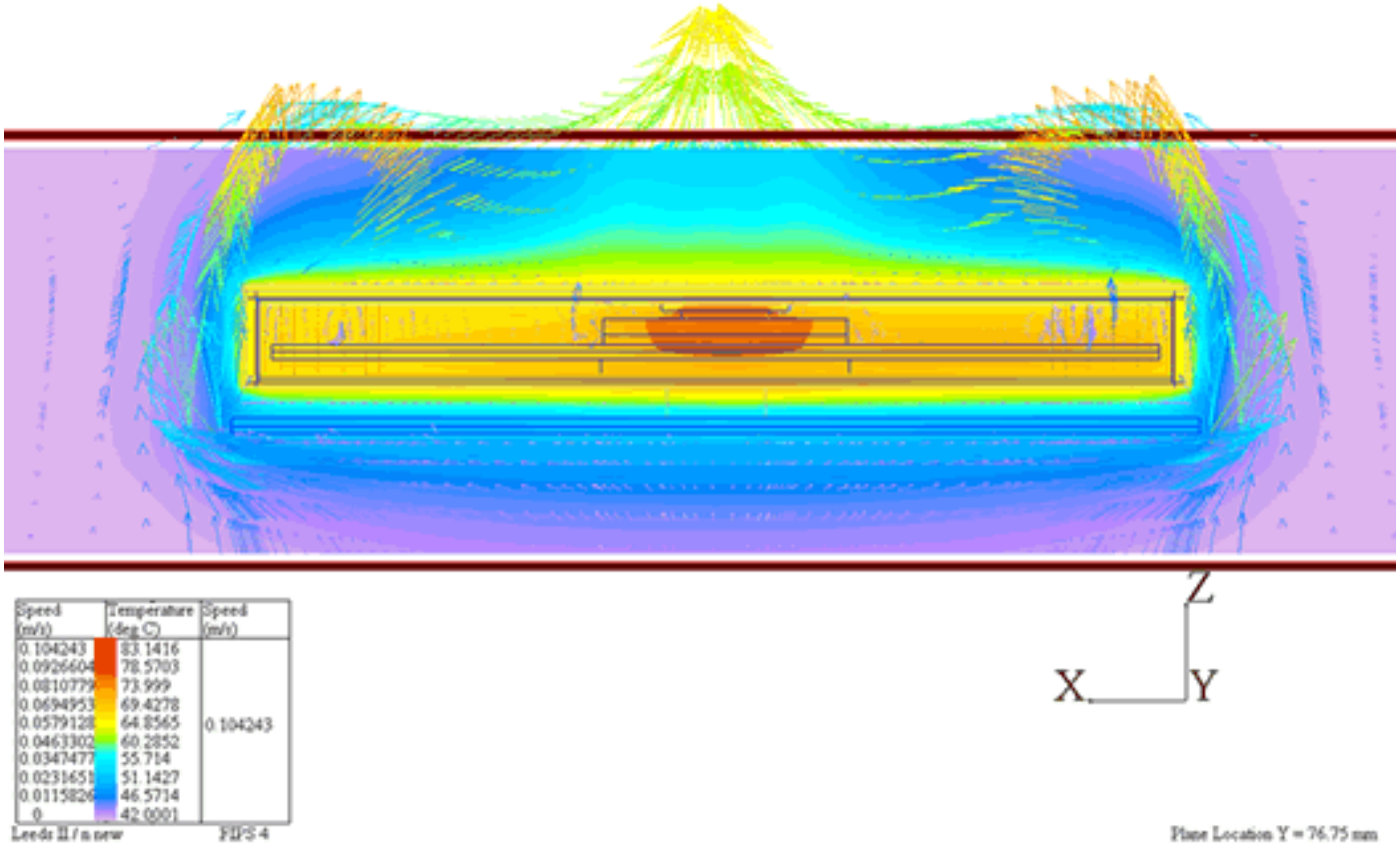


(2D model, 2.5 D exists)

http://lava.cs.virginia.edu/HotSpot/documentation.htm
Including PowerPoint slides from ISCA 2003

Results of simulations based on thermal models (1)

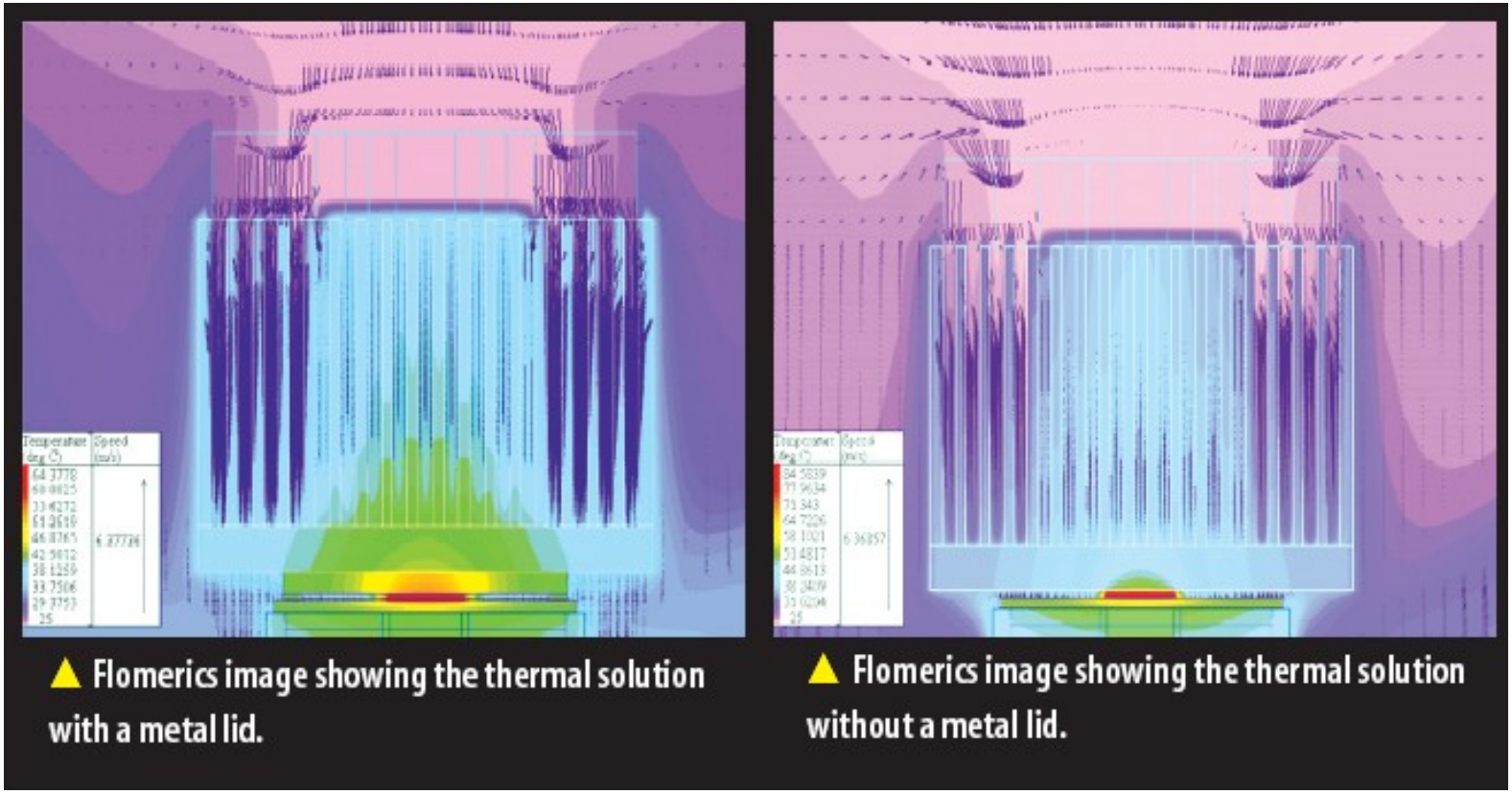
Encapsulated cryptographic coprocessor:



Source: http://www.coolingzone.com/Guest/News/NL_JUN_2001/Campi/Jun_Campi_2001.html

Results of simulations based on thermal models (2)

Microprocessor



▲ Flomerics image showing the thermal solution with a metal lid.

▲ Flomerics image showing the thermal solution without a metal lid.

Source: http://www.flotherm.com/applications/app141/hot_chip.pdf

Summary

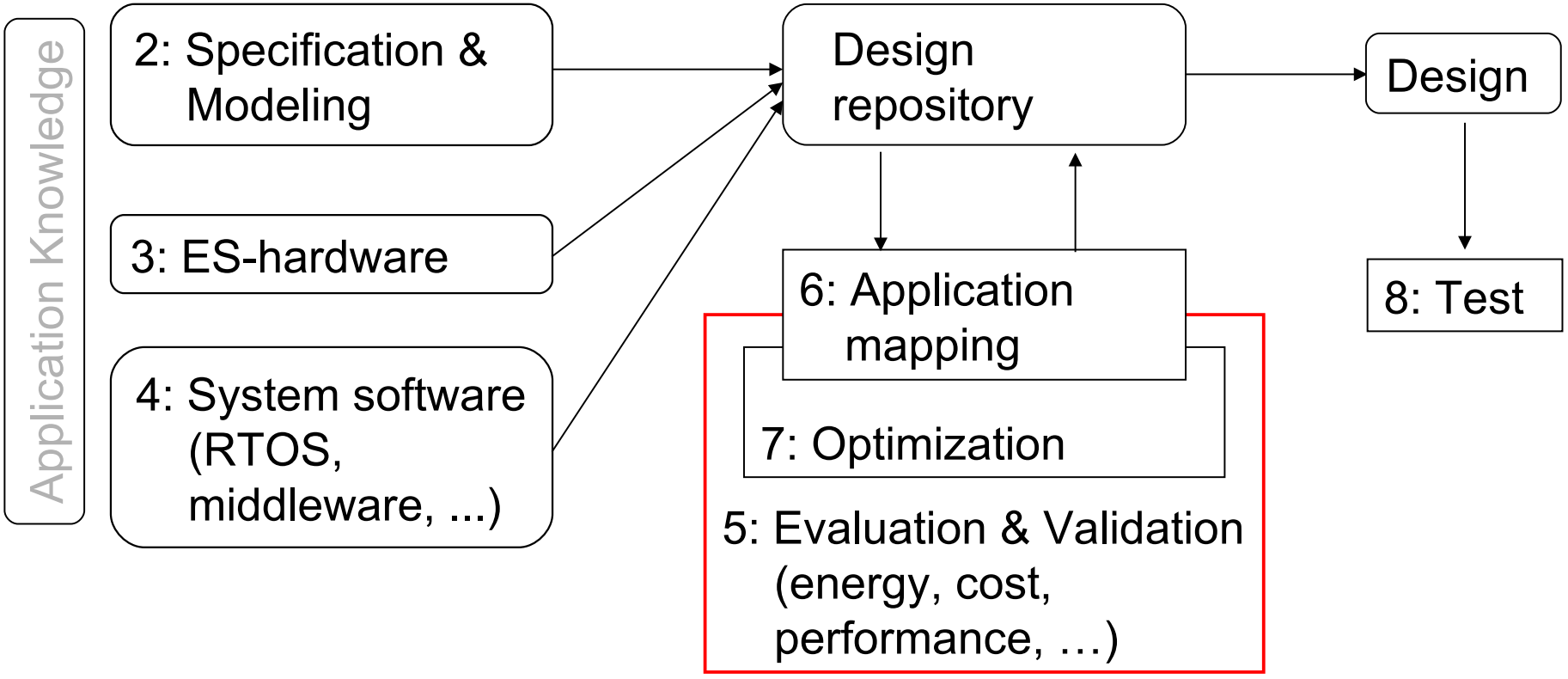
- Thiele's real-time calculus (RTC)/MPA
  - Using bounds on the number of events in input streams
  - Using bounds on available processing capability
  - Derives bounds on the number of events in output streams
  - Derives bound on remaining processing capability, buffer sizes, …
  - Examples demonstrate design procedure based on RTC
- Energy and power consumption
  - Measurements
  - Models (with calibration)
  - Mixed approaches
  - Energy for computation, storage and communication
- Thermal behavior
  - Mapping to thermal circuit model

Evaluation and Validation

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 12 月 11 日

Structure of this course

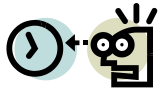


Numbers denote sequence of chapters

How to evaluate designs according to multiple criteria?

Many different criteria are relevant for evaluating designs:

- Average & worst case delay
- power/energy consumption
- thermal behavior
- reliability, safety, security
- cost, size
- weight
- EMC characteristics
- radiation hardness, environmental friendliness, ..









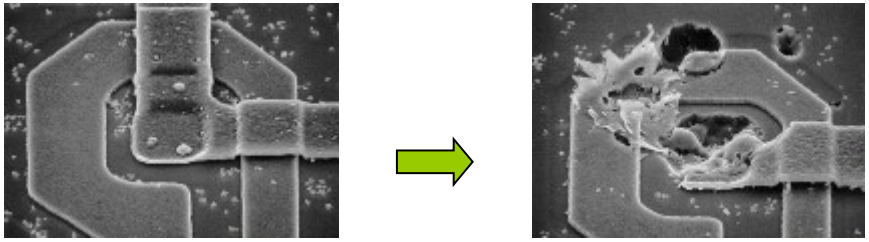
How to compare different designs?
(Some designs are “better” than others)

Impact of shrinking feature sizes

- Reduced reliability due to smaller patterns within semiconductor chips [ITRS, 2009]
- Transient & permanent faults
- Several types of faults
  Example: Electro-migration

metal migration
@ Pentium 4

www.jrwhipple.com/computer_hangs.html



- Rate of faults expected to increase such that designs need to become fault-tolerant

Terms

- "*A **service failure**, often abbreviated here to **failure**, is an event that occurs when the delivered service of a system deviates from the correct service.*"
- *"The definition of an **error** is the part of the total state of the system that may lead to its subsequent service failure".*
- "*The adjudged or hypothesized cause of an error is called a **fault**. Faults can be internal or external of a system.*"

Example:

- Transient **fault** flipping a bit in memory.
- After this bit flip, the memory cell will be in **error**.
- **Failure***:* if the system service is affected by this error.

We will consider **failure** rates & **fault** models.

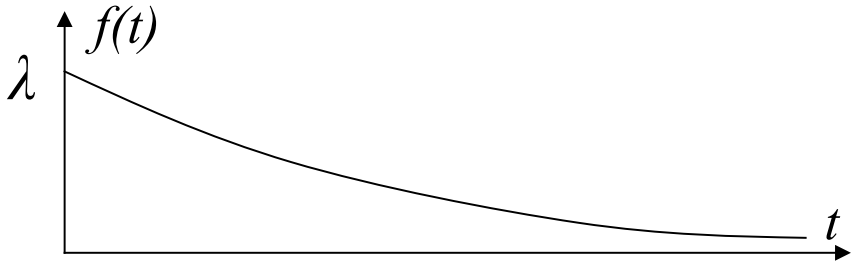
[Laprie et al., 1992, 2004]

Reliability: *f(t)*, *F(t)*

- Let T: time until first failure (random variable)
- Let $f(t)$ be the density function of T

Example: Exponential distribution

$$f(t) = \lambda e^{-\lambda t}$$

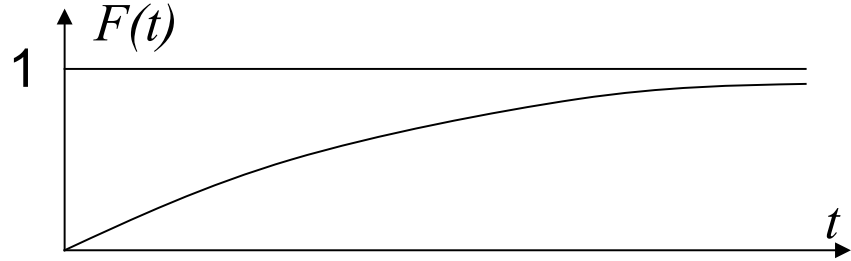


- $F(t)$ = probability of the system being faulty at time t:

$$F(t) = Pr(T \le t) \qquad F(t) = \int_0^t f(x)\,dx$$

Example: Exponential distribution

$$F(t) = \int_0^t \lambda e^{-\lambda x}\,dx = -\left[e^{-\lambda x}\right]_0^t = 1 - e^{-\lambda t}$$



Reliability: *R(t)*

- **Reliability** $R(t)$ = probability that the time until the first failure is larger than some time t:

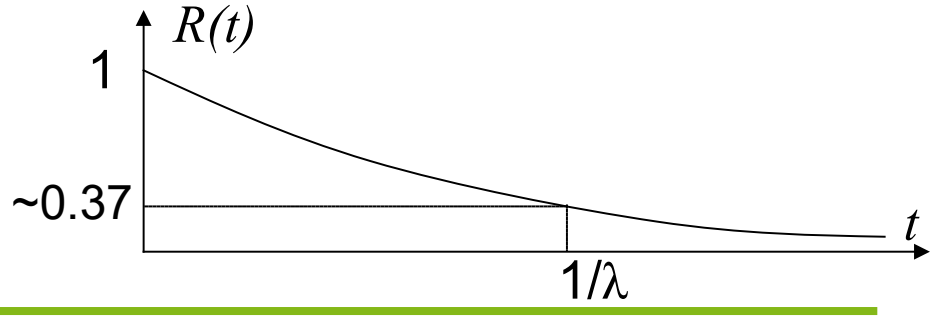
$$R(t) = Pr(T > t), t \geq 0 \qquad R(t) = \int_t^{\infty} f(x)\,dx$$

$$F(t) + R(t) = \int_0^t f(x)\,dx + \int_t^{\infty} f(x)\,dx = 1$$

$$R(t) = 1 - F(t) \qquad f(t) = \frac{-dR(t)}{dt}$$

Example: Exponential distribution

$$R(t) = e^{-\lambda t}$$



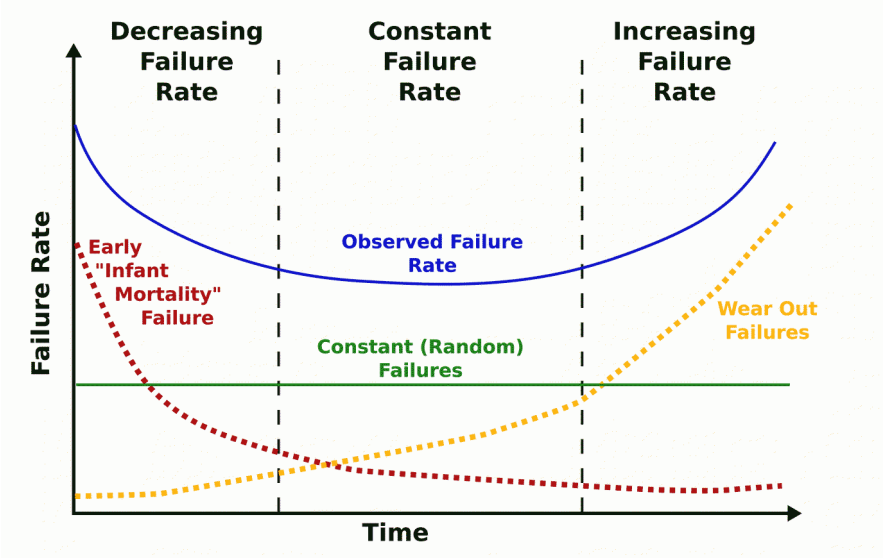
Failure rate

The failure rate at time t is the probability of the system failing between time t and time $t+\Delta t$:

$$\lambda(t) = \lim_{\Delta t \to 0} \frac{Pr(t < T \leq t + \Delta t \mid T > t)}{\Delta t} = \lim_{\Delta t \to 0} \frac{F(t+\Delta t) - F(t)}{\Delta t\, R(t)} = \frac{f(t)}{R(t)}$$

Conditional probability ("provided that the system works at t ");

$Pr(A|B) = Pr(AB)/Pr(B)$



For exponential distribution:

$$\frac{f(t)}{R(t)} = \frac{\lambda e^{-\lambda t}}{e^{-\lambda t}} = \lambda$$

FIT = expected number of failures in 10^9 hrs.

FIT & “10^{-9}“ (per hour)

“10^{-9}“: For many systems, probability of a catastrophe has to be less than 10^{-9} per hour ≡ one case per 100,000 systems for 10,000 hours.

FIT: failure-in-time unit for failure rate

1 FIT: rate of 10^{-9} failures per hour

MTTF = $E\{T\}$, the *statistical mean* value of T

$$\text{MTTF} = E\{T\} = \int_0^\infty t \cdot f(t)\, dt$$

According to the definition of the statistical mean value

Example: Exponential distribution

$$\text{MTTF}_{exp} = \int_0^\infty t \cdot \lambda e^{-\lambda t}\, dt = -\left[t \cdot e^{-\lambda t}\right]_0^\infty + \int_0^\infty e^{-\lambda t}\, dt \qquad \int u \cdot v' = u \cdot v - \int u' \cdot v$$

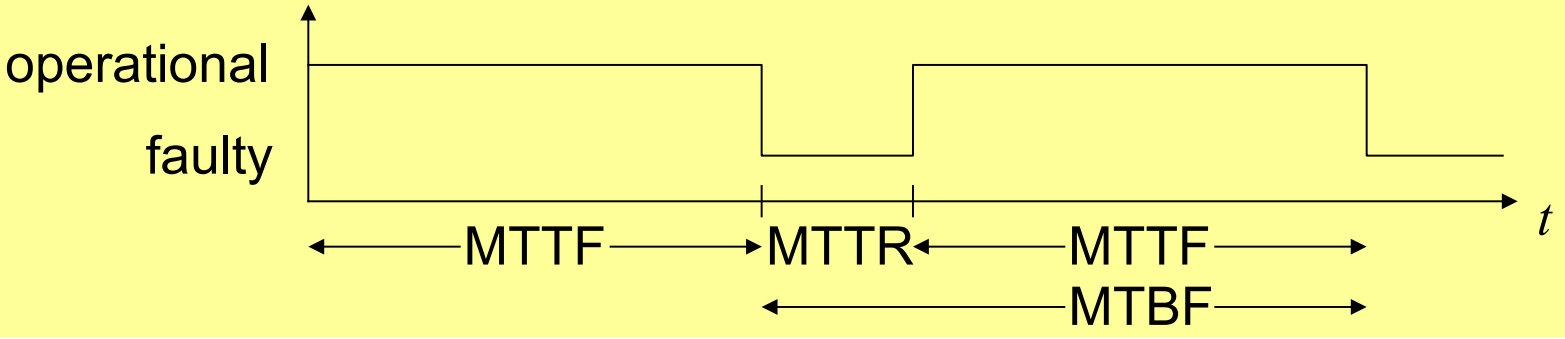
$$\text{MTTF}_{exp} = \frac{-1}{\lambda}\left[e^{-\lambda t}\right]_0^\infty = \frac{-1}{\lambda}\left[0 - 1\right] = \frac{1}{\lambda}$$

MTTF is the reciprocal value of failure rate.

MTTF, MTTR and MTBF

MTTR = mean time to repair
(average over repair times using distribution $M(d)$)
MTBF* = mean time between failures = MTTF + MTTR

Ignoring the statistical nature of failures …



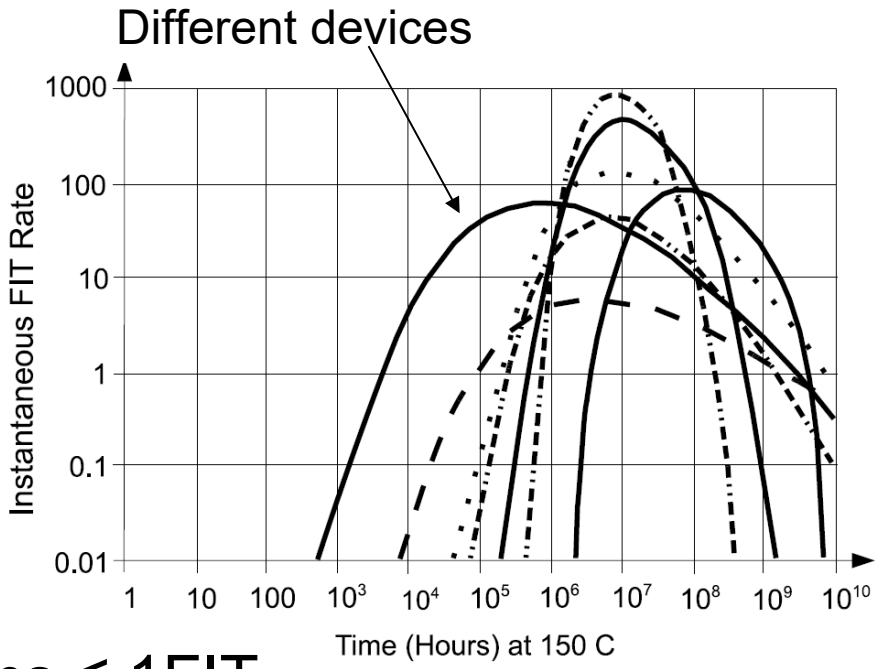
Availability $A = \lim_{t \to \infty} A(t) = \frac{MTTF}{MTBF}$

* Mixed up with MTTF, if starting in operational state is implicitly assumed

Actual failure rates

Failure rates derived from experiments at higher temperatures.

Example: failure rates less than 100 FIT for the first 20 years (175,300 hrs) of life at 150°C @ TriQuint (GaAs)
www.triquint.com/company/quality/faqs/faq_11.cfm



Target: Failures rates of systems ≤ 1FIT

Reality: Failures rates of circuits ≤ 100 FIT

☞ redundancy is required to make a system more reliable than its components

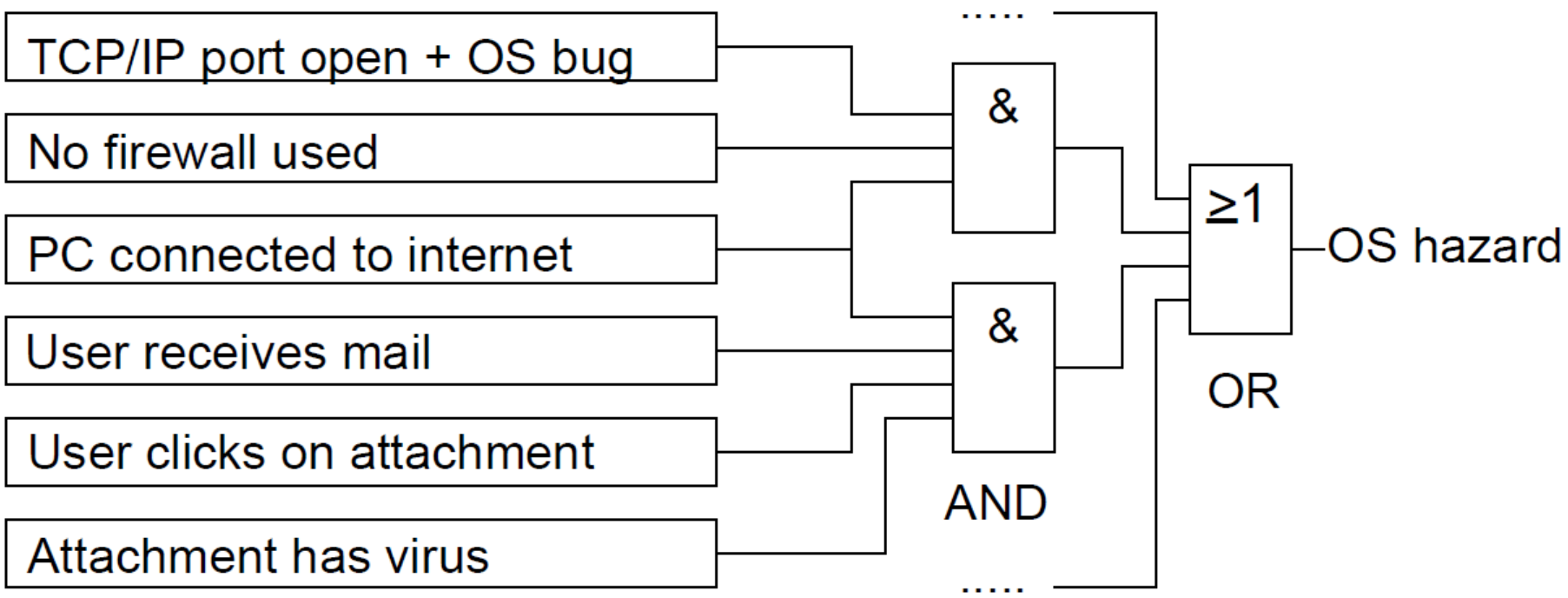
∃ non-constant failure rates!

Fault Tree Analysis (FTA)

Damages are resulting from hazards/risks.
For every damage there is a severity and a probability.
Several techniques for analyzing risks.

- FTA is a top-down method of analyzing risks. Analysis starts with possible damage, tries to come up with possible scenarios that lead to that damage.
- FTA typically uses a graphical representation of possible damages, including symbols for AND- and OR-gates.
- OR-gates are used if a single event could result in a hazard.
- AND-gates are used when several events or conditions are required for that hazard to exist.

Example


TCP/IP port open + OS bug
No firewall used
PC connected to internet
User receives mail
User clicks on attachment
Attachment has virus
.....
&
&
AND
.....
≥1
OR
OS hazard

Limitations

The simple AND- and OR-gates cannot model all situations.

For example, their modeling power is exceeded if shared resources of some limited amount (like energy or storage locations) exist.

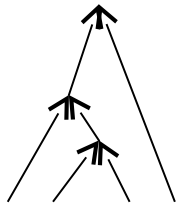
Markov models may have to be used to cover such cases.

Failure mode and effect analysis (FMEA)

- FMEA starts at the components and tries to estimate their reliability. The first step is to create a table containing components, possible faults, probability of faults and consequences on the system behavior.

Component	*Failure*	*Consequences*	*Probability*	*Critical?*
...	...	...	...	...
Processor	metal migration	no service	10^{-7} /h	yes
...	...	...	...	...

- Using this information, the reliability of the system is computed from the reliability of its parts (corresponding to a bottom-up analysis).



Safety cases

- Both approaches (FTA & FMEA) may be used in *“safety cases”*.
- In such cases, an independent authority has to be convinced that certain technical equipment is indeed safe.
- One of the commonly requested properties of technical systems is that no single failing component should potentially cause a catastrophe.

Dependability requirements

Allowed failures may be in the order of 1 failure per 10^9 h.

~ 1000 times less than typical failure rates of chips.

☞ For safety-critical systems, the system as a whole must be more dependable than any of its parts.

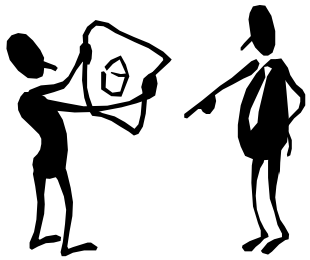
☞ fault-tolerance mechanisms must be used.

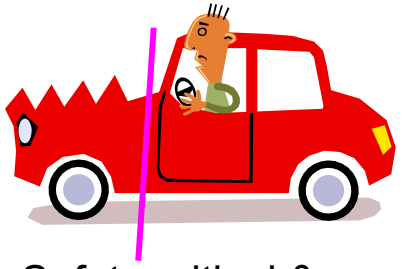
Low acceptable failure rate → systems not 100% testable.

☞ Safety must be shown by a combination of testing and reasoning. Abstraction must be used to make the system explainable using a hierarchical set of behavioral models. Design faults and human failures must be taken into account.

Kopetz's 12 design principles (1-3)

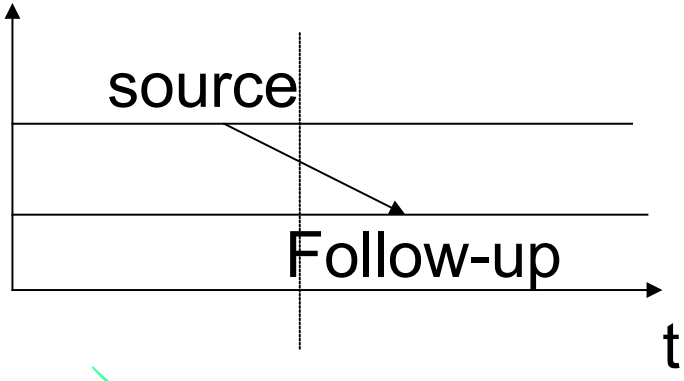
1. Safety considerations may have to be used as the important part of the specification, driving the entire design process.
2. Precise specifications of design hypotheses must be made right at the beginning. These include expected failures and their probability.
3. Fault containment regions (FCRs) must be considered. Faults in one FCR should not affect other FCRs.

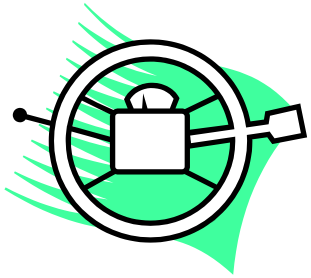


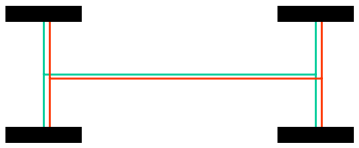


Kopetz's 12 design principles (4-6)

4. A consistent notion of time and state must be established. Otherwise, it will be impossible to differentiate between original and follow-up errors.
5. Well-defined interfaces have to hide the internals of components.
6. It must be ensured that components fail independently.







2 independent brake hose systems

Kopetz's 12 design principles (7-9)

7. Components should consider themselves to be correct unless two or more other components pretend the contrary to be true (principle of self-confidence).

one of the systems sufficient for braking

8. Fault tolerance mechanisms must be designed such that they do not create any additional difficulty in explaining the behavior of the system. Fault tolerance mechanisms should be decoupled from the regular function.

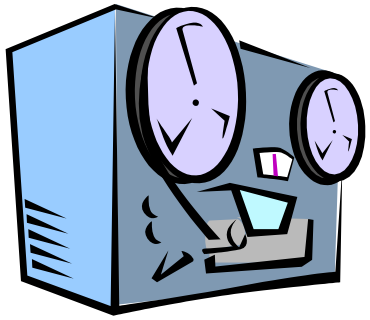
9. The system must be designed for diagnosis. For example, it has to be possible to identifying existing (but masked) errors.

Kopetz's 12 design principles (10-12)

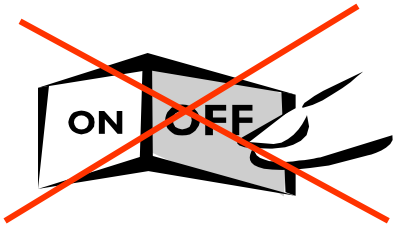
10. The man-machine interface must be intuitive and forgiving. Safety should be maintained despite mistakes made by humans.

airbag

11. Every anomaly should be recorded. These anomalies may be unobservable at the regular interface level. Recording to involve internal effects, otherwise they may be masked by fault-tolerance mechanisms.



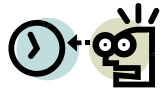
12. Provide a never-give up strategy. ES may have to provide uninterrupted service. Going offline is unacceptable.

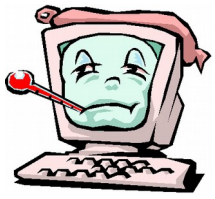


How to evaluate designs according to multiple criteria?

Many different criteria are relevant for evaluating designs:

- Average & worst case delay
- power/energy consumption
- thermal behavior
- reliability, safety, security
- cost, size
- weight
- EMC characteristics
- radiation hardness, environmental friendliness, ..

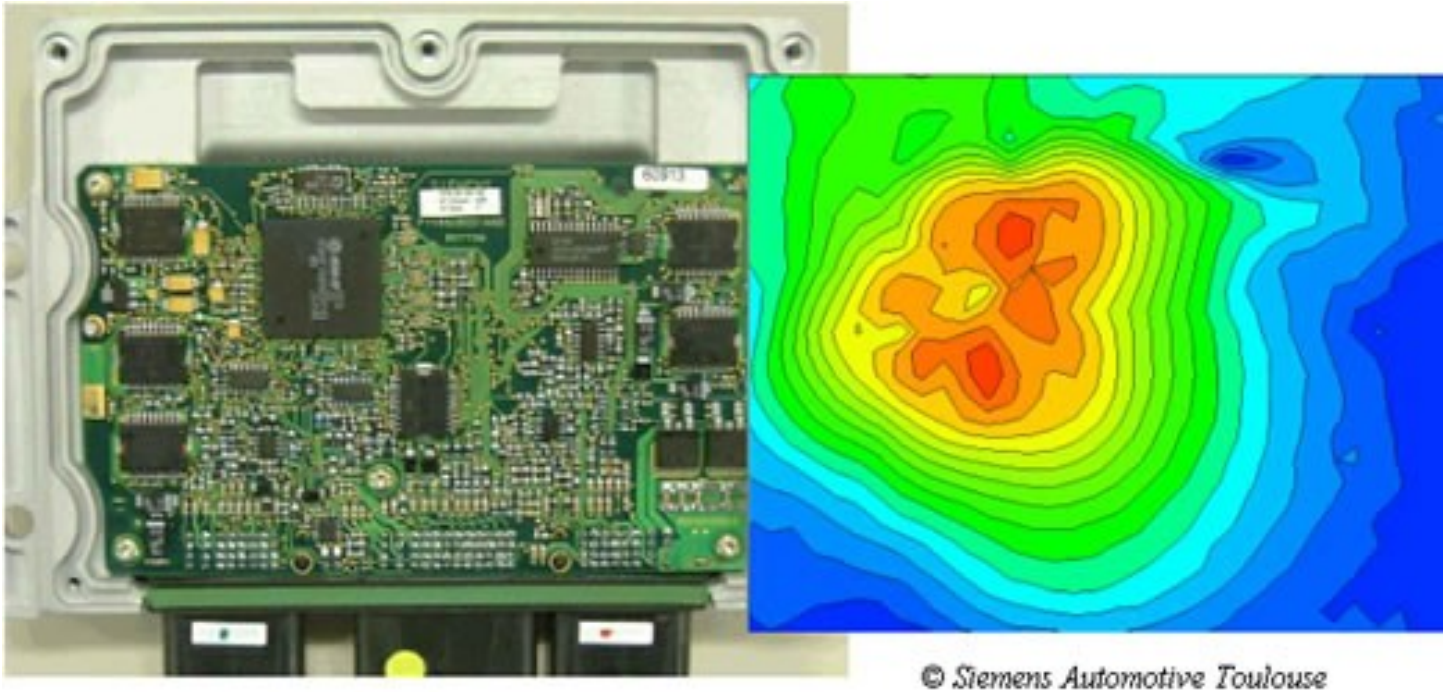





How to compare different designs?
(Some designs are “better” than others)

Electro-magnetic compatibility (EMC)

Example: car engine controller



© Siemens Automotive Toulouse

Red: high emission; Validation of EMC properties often done at the end of the design phase.

http://intrage.insa-tlse.fr/~etienne/emccourse/what_for.html

Simulations

- Simulations try to imitate the behavior of the real system on a (typically digital) computer.
- Simulation of the functional behavior requires executable models.
- Simulations can be performed at various levels.
- Some non-functional properties (e.g. temperatures, EMC) can also be simulated.
- Simulations can be used to **evaluate** and to **validate** a design

Validating functional behavior by simulation

Various levels of abstractions used for simulations:

- High-level of abstraction: fast, but sometimes not accurate
- Lower level of abstraction: slow and typically accurate
- Choosing a level is always a compromise

Simulations: Limitations

- Typically slower than the actual design.
  ☞ **Violations of timing constraints** likely if simulator is connected to the actual environment

- Simulations in the real environment may be **dangerous**

- There may be huge amounts of data and it may be impossible to simulate enough data in the available time.

- Most actual systems are too complex to allow simulating all possible cases (inputs).
  Simulations can help finding errors in designs, but they cannot guarantee the absence of errors.

Rapid prototyping/Emulation

- Prototype: Embedded system that can be generated quickly and behaves very similar to the final product.
- May be larger, more power consuming and have other properties that can be accepted in the validation phase
- Can be built, for example, using FPGAs.

Example:
Quickturn Cobalt System (1997),
~0.5M$ for 500kgate entry level system

http://www.eedesign.com/editorial/1997/toolsandtech9703.html

Emulation

- Simulations: based on models, which are approximations of real systems.
- In general: ∃ difference between real system and model.
- Reduce gap by implementing parts of SUD more precisely!

Definition: Emulation is the process of executing a model of the SUD where at least one component is **not** represented by simulation on some kind of host computer.

"*Bridging the credibility gap is not the only reason for a growing interest in emulation—the above definition of an emulation model remains valid when turned around— an emulation model is one where part of the real system is replaced by a model. Using emulation models to test control systems under realistic conditions, by replacing the* 'real system' *with a model, is proving to be of considerable interest …"* [McGregor, 2002]

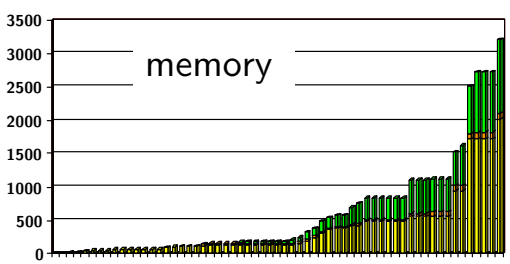
Probleme (cont.)

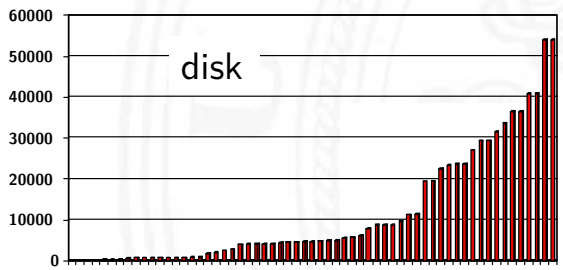
- Entwurfswerkzeuge / Hardwareaufwand
  - EDA-Werkzeuge + Hardware

Quelle: NVIDIA
DAC'02

The Investment

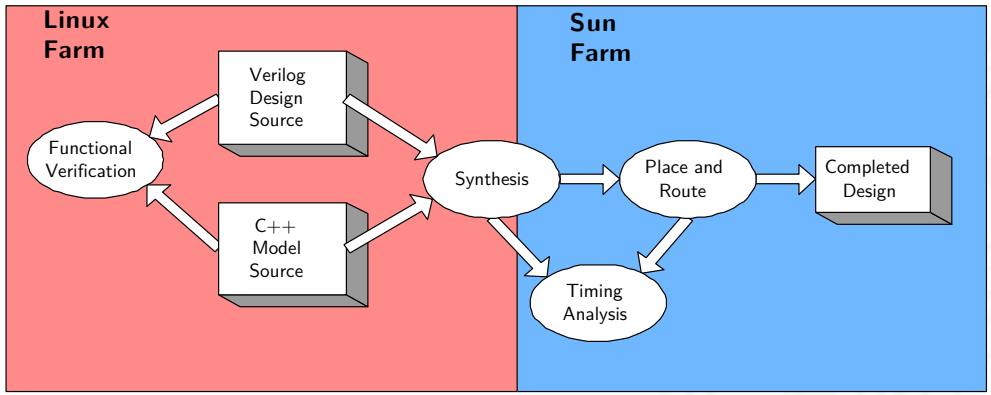
- ~$160M of CAD tools installed and online
- ~$40M Emulation installation
- Engineering Compute resources
  - Desktops: 225 Suns (Solaris), 5000 x86 PCs (Linux/NT)
  - Server CPUs: 450 Sparc (Solaris), 2700 x86 (Linux/NT)
  - 3.2 Terabytes RAM, 55 Terabytes of storage

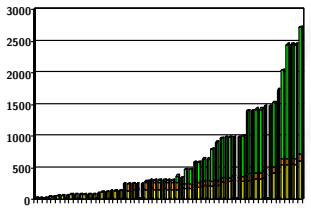




Probleme (cont.)

Compute Resource Allocation





Probleme (cont.)

- Hardware Emulation $>$ 100 Mio. \$ DAC'06

Example of a recent commercial emulator

www.verisity.com/images/products/xtremep{1|3}.gif

Formal verification

- Formal verification = formally proving a system correct, using the language of mathematics.
- Formal model required. Obtaining this cannot be automated.
- Model available ☞ try to prove properties.
- Even a formally verified system can fail (e.g. if assumptions are not met).
- Classification by the type of logics.

Ideally: Formally verified tools transforming specifications into implementations ("*correctness by construction*").

In practice: Non-verified tools and manual design steps ☞ validation of each and every design required. Unfortunately has to be done at intermediate steps and not just for the final design ☞ Major effort required.

Propositional logic (1)

- Consisting of Boolean formulas comprising Boolean variables and connectives such as ∨ and ∧.
- Gate-level logic networks can be described.
- Typical aim: checking if two models are equivalent (called **tautology checkers** or **equivalence checkers)**.
- Since propositional logic is decidable, it is also decidable whether or not the two representations are equivalent.
- Tautology checkers can frequently cope with designs which are too large to allow simulation-based exhaustive validation.

Propositional logic (2)

- Reason for power of tautology checkers: Binary Decision Diagrams (BDDs)
- Complexity of equivalence checks of Boolean functions represented with BDDs: *O*(*number of BDD-nodes*) (equivalence check for sums of products is NP-hard). #(BDD-nodes) not to be ignored!
- Many functions can be efficiently represented with BDDs. In general, however, the #(nodes) of BDDs grows exponentially with the number of variables.
- Simulators frequently replaced by equivalence checkers if functions can be efficiently represented with BDDs.
- Very much limited ability to verify FSMs.

First order logic (FOL)

- FOL includes quantification, using $\exists$ and $\forall$.
- Some automation for verifying FOL models is feasible.
- However, since FOL is undecidable in general, there may be cases of doubt.

Higher order logic (HOL)

- Higher order logic allows functions to be manipulated like other objects.
- For higher order logic, proofs can hardly ever be automated and typically must be done manually with some proof-support.

Model checking

- Aims at the verification of finite state systems.
- Analyzes the state space of the system.
- Verification using this approach requires three stages:
  - generation of a model of the system to be verified,
  - definition of the properties expected, and
  - model checking (the actual verification step).

Computational properties

- Model checking is easier to automate than FOL.
- In 1987, model checking was implemented using BDDs.
- It was possible to locate several errors in the specification of the *future bus* protocol.
- Model checking becoming very popular
- Extensions are needed in order to also cover real-time behavior and numbers.

Summary

Evaluation and Validation:

- Reliability
  - Definitions
  - Failure rates
  - MTBF, MTTF, MTTR
  - Fault tree analysis, FMEA
  - Kopetz' 12 principles
- Electro-magnetic compatibility (briefly)
- Simulation, Emulation
- Formal verification
  - Propositional,
  - first order, higher order based techniques,
  - model checking

technische universität
dortmund

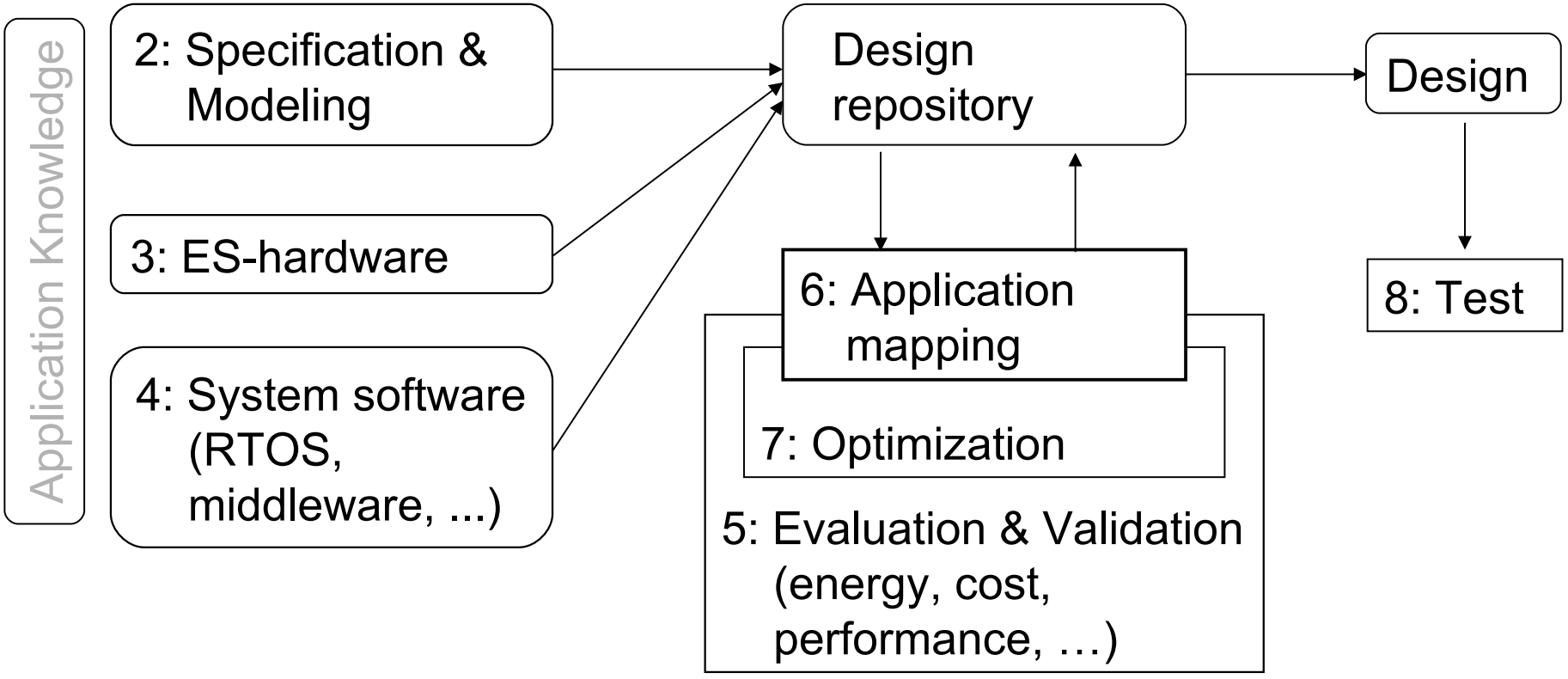
fakultät für informatik
informatik 12

Mapping of Applications to Platforms

Peter Marwedel
TU Dortmund,
Informatik 12

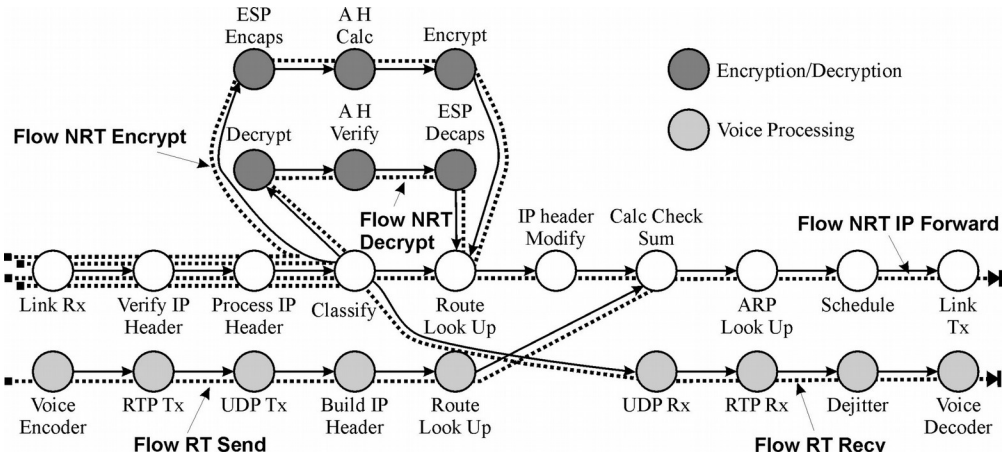
2012 年 12 月 12 日

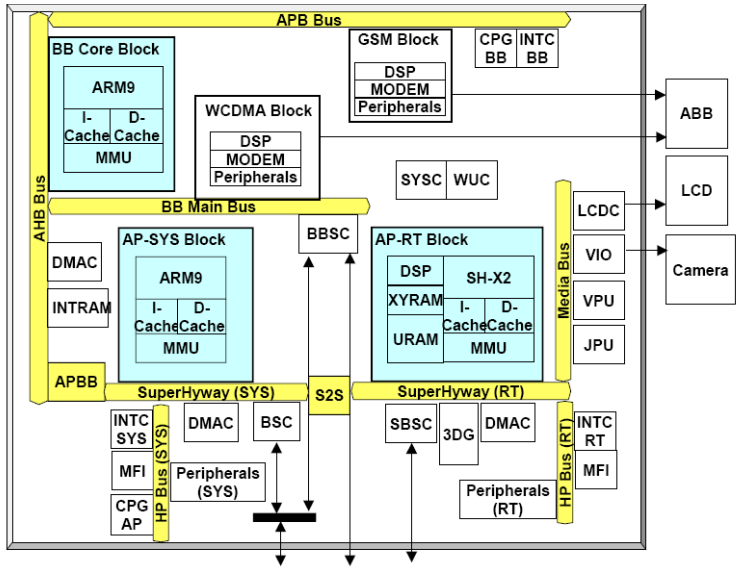
Structure of this course



Numbers denote sequence of chapters

Mapping of Applications to Platforms





Distinction between mapping problems

	Embedded	PC-like
Architectures	Frequently heterogeneous very compact	Mostly homogeneous not compact (x86 etc)
x86 compatibility	Less relevant	Very relevant
Architecture fixed?	Sometimes not	Yes
Model of computation (MoCs)	C+multiple models (data flow, discrete events, ...)	Mostly von Neumann (C, C++, Java)
Optim. objectives	Multiple (energy, size, ...)	Average performance dominates
Real-time relevant	Yes, very!	Hardly
Applications	Several concurrent apps.	Mostly single application
Apps. known at design time	Most, if not all	Only some (e.g. WORD)

Problem Description

Given

- A set of applications
- Scenarios on how these applications will be used
- A set of candidate architectures comprising
  - (Possibly heterogeneous) processors
  - (Possibly heterogeneous) communication architectures
  - Possible scheduling policies

Tools urgently needed!

Find

- A mapping of applications to processors
- Appropriate scheduling techniques (if not fixed)
- A target architecture (if DSE is included)

= Design Space Exploration

Objectives

- Keeping deadlines and/or maximizing performance
- Minimizing cost, energy consumption

Related Work

- Mapping to EXUs in automotive design
- Scheduling theory:
  Provides insight for the mapping *task* → *start times*
- Hardware/software partitioning:
  Can be applied if it supports multiple processors
- High performance computing (HPC)
  Automatic parallelization, but only for
  - single applications,
  - fixed architectures,
  - no support for scheduling,
  - memory and communication model usually different
- High-level synthesis
  Provides useful terms like scheduling, allocation, assignment
- Optimization theory

Scope of mapping algorithms

Useful terms from hardware synthesis:

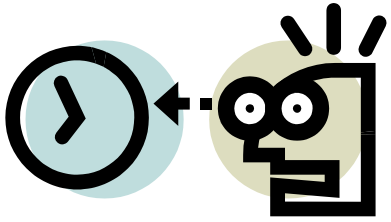
- **Resource Allocation**
  Decision concerning type and number of available resources
- **Resource Assignment**
  Mapping: Task → (Hardware) Resource
- **xx to yy binding:**
  Describes a mapping from behavioral to structural domain, e.g. task to processor binding, variable to memory binding
- **Scheduling**
  Mapping: Tasks → Task start times
  Sometimes, resource assignment is considered being included in scheduling.

Classes of mapping algorithms considered in this course

- **Classical scheduling algorithms**
  Mostly for independent tasks & ignoring communication, mostly for mono- and homogeneous multiprocessors
- **Dependent tasks as considered in architectural synthesis**
  Initially designed in different context, but applicable
- **Hardware/software partitioning**
  Dependent tasks, heterogeneous systems, focus on resource assignment
- **Design space exploration using evolutionary algorithms**
  Heterogeneous systems, incl. communication modeling

Real-time scheduling

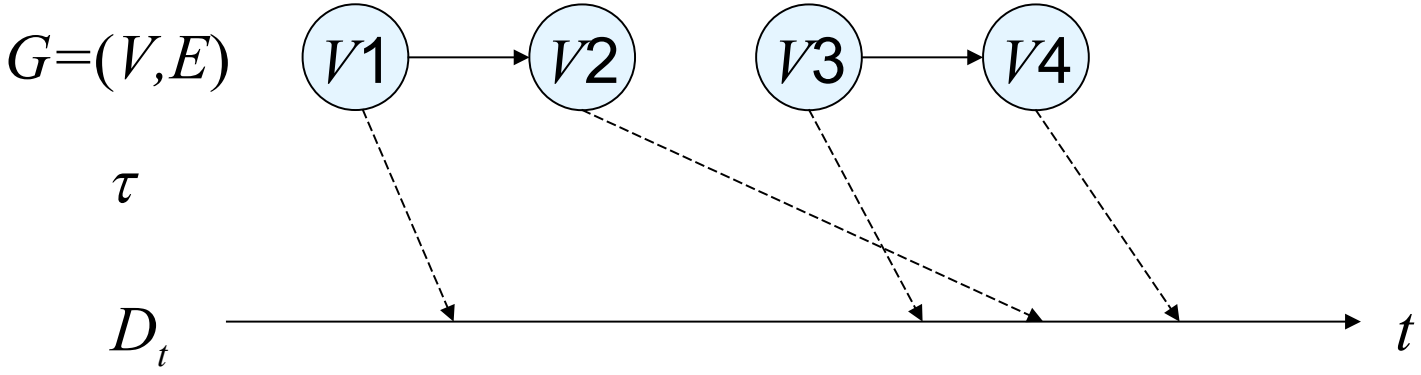
Assume that we are given a task graph $G=(V,E)$.



Def.: A **schedule** τ of G is a mapping

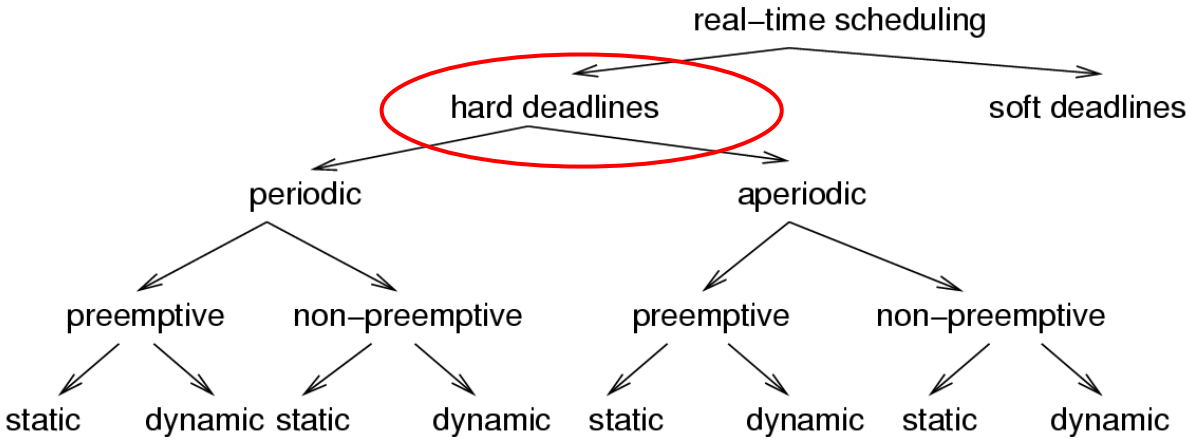
$$V \rightarrow D_t$$

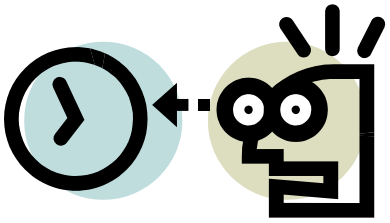
of a set of tasks V to start times from domain D_t.



Typically, schedules have to respect a number of constraints, incl. resource constraints, dependency constraints, deadlines.
Scheduling = finding such a mapping.

Hard and soft deadlines



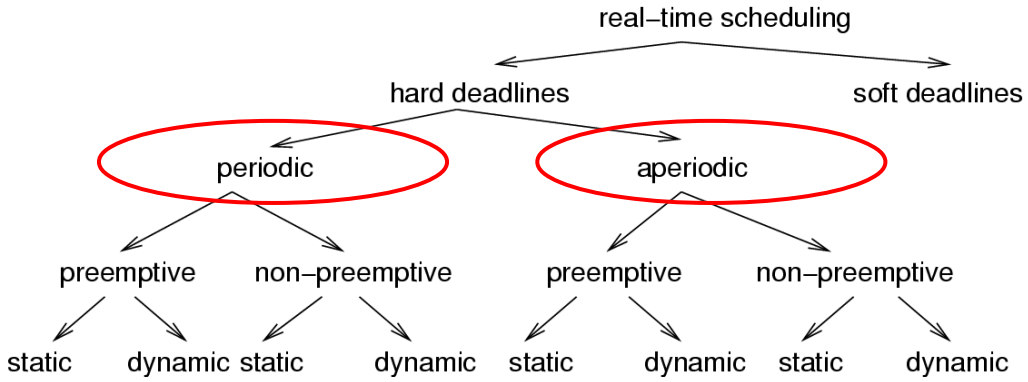


Def.: A time-constraint (deadline) is called **hard** if not meeting that constraint could result in a catastrophe [Kopetz, 1997].

All other time constraints are called **soft**.

We will focus on hard deadlines.

Periodic and aperiodic tasks

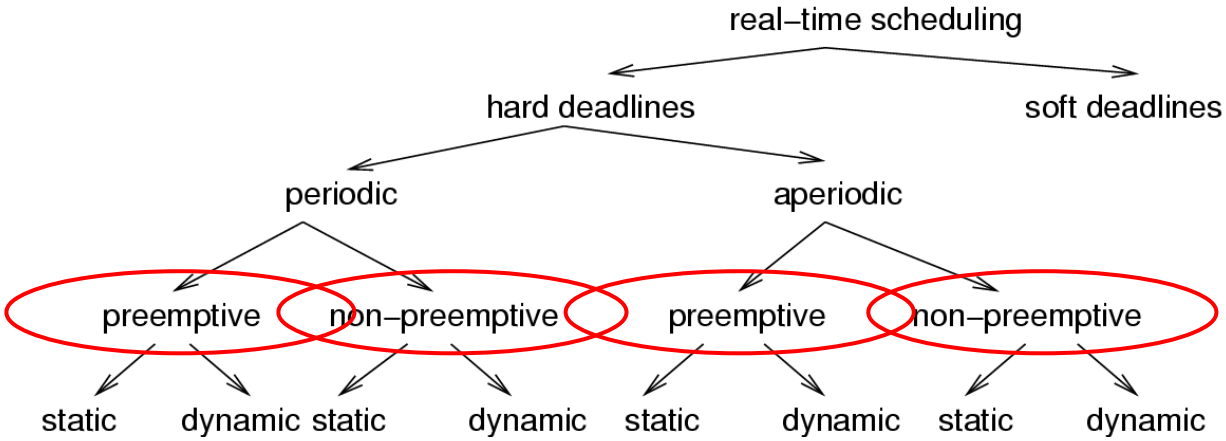


Def.: Tasks which must be executed once every *p* units of time are called **periodic** tasks. *p* is called their period. Each execution of a periodic task is called a **job**.

All other tasks are called **aperiodic**.

Def.: Tasks requesting the processor at unpredictable times are called **sporadic**, if there is a minimum separation between the times at which they request the processor.

Preemptive and non-preemptive scheduling



- **Non-preemptive schedulers:**

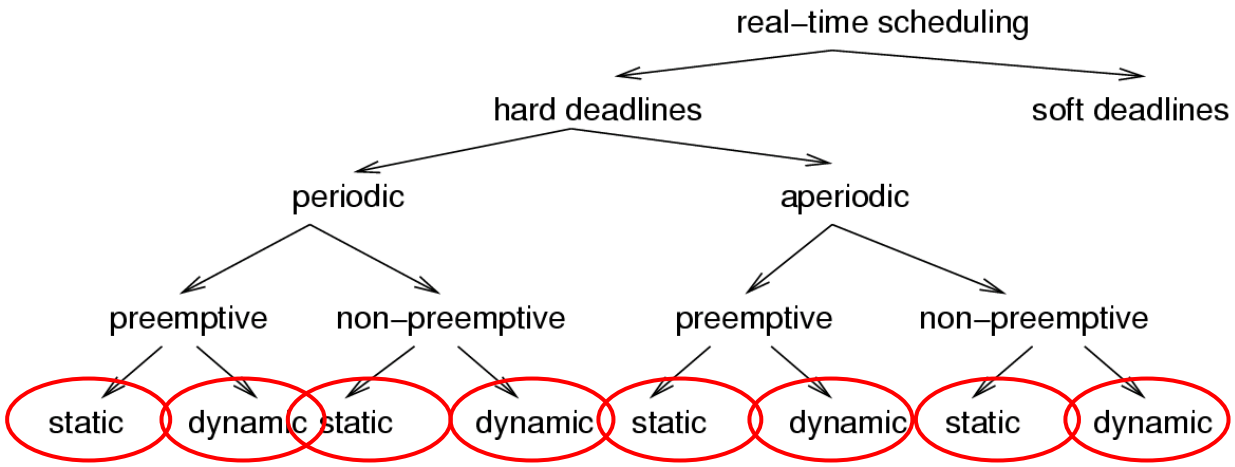
  Tasks are executed until they are done.

  Response time for external events may be quite long.

- **Preemptive schedulers:** To be used if
  - some tasks have long execution times or
  - if the response time for external events to be short.

Dynamic/online scheduling

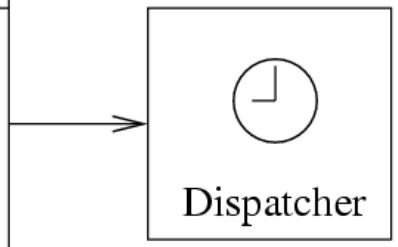
- **Dynamic/online scheduling:**
  Processor allocation decisions (scheduling) at run-time; based on the information about the tasks arrived so far.



Static/offline scheduling

- **Static/offline scheduling:**
  Scheduling taking a priori knowledge about arrival times, execution times, and deadlines into account.
  Dispatcher allocates processor when interrupted by timer.
  Timer controlled by a table generated at design time.

Time	Action	WCET
10	start T1	12
17	send M5	
22	stop T1	
38	start T2	20
47	send M3	

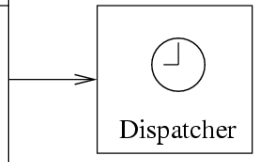


Time-triggered systems (1)

In an entirely time-triggered system, the temporal control structure of all tasks is established ***a priori*** *by off-line support-tools. This temporal control structure is encoded in a* ***Task-Descriptor List (TDL)*** *that contains the cyclic schedule for all activities of the node. This schedule considers the required precedence and mutual exclusion relationships among the tasks such that an explicit coordination of the tasks by the operating system at run time is not necessary. ..*

The dispatcher is activated by the synchronized clock tick. It looks at the TDL, and then performs the action that has been planned for this instant [Kopetz].

Time	Action	WCET
10	start T1	12
17	send M5	
22	stop T1	
38	start T2	20
47	send M3	



Time-triggered systems (2)

… pre-run-time scheduling is often the only practical means of providing predictability in a complex system. [Xu, Parnas].

- It can be easily checked if timing constraints are met.
- The disadvantage is that the response to sporadic events may be poor.

Centralized and distributed scheduling

- **Mono- and multi-processor scheduling:**
  - Simple scheduling algorithms handle single processors,
  - more complex algorithms handle multiple processors.
    - algorithms for homogeneous multi-processor systems
    - algorithms for heterogeneous multi-processor systems (includes HW accelerators as special case).

- **Centralized and distributed scheduling:** Multiprocessor scheduling either locally on 1 or on several processors.

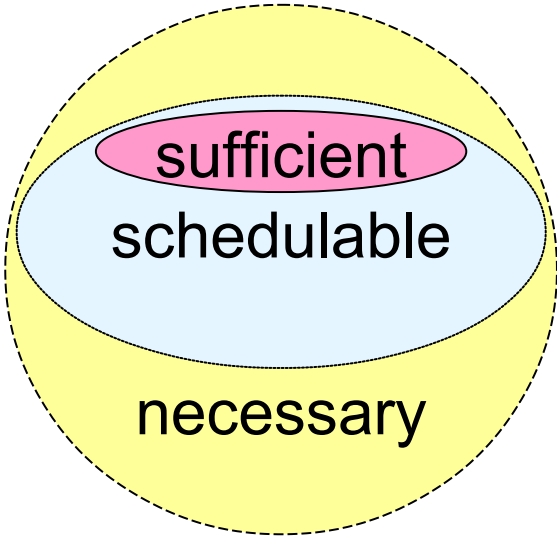
Schedulability

Set of tasks is **schedulable** under a set of constraints, if a schedule exists for that set of tasks & constraints.

Exact tests are NP-hard in many situations.

Sufficient tests: sufficient conditions for schedule checked. (Hopefully) small probability of not guaranteeing a schedule even though one exists.

Necessary tests: checking necessary conditions. Used to show no schedule exists. There may be cases in which no schedule exists & we cannot prove it.



Cost functions

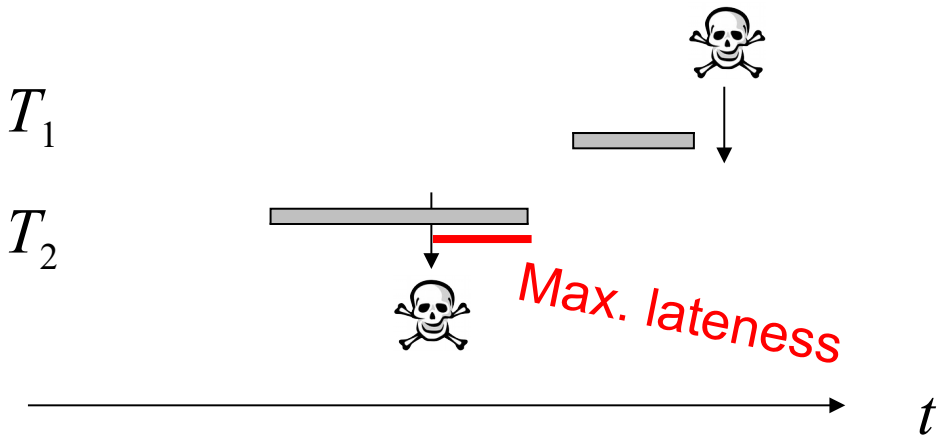
Cost function:

Different algorithms aim at minimizing different functions.

Def.:

Maximum lateness = $\max_{\text{all tasks}}$ (completion time – deadline)

Is <0 if all tasks complete before deadline.



technische universität
dortmund

fakultät für informatik
informatik 12

Classical scheduling algorithms for aperiodic systems

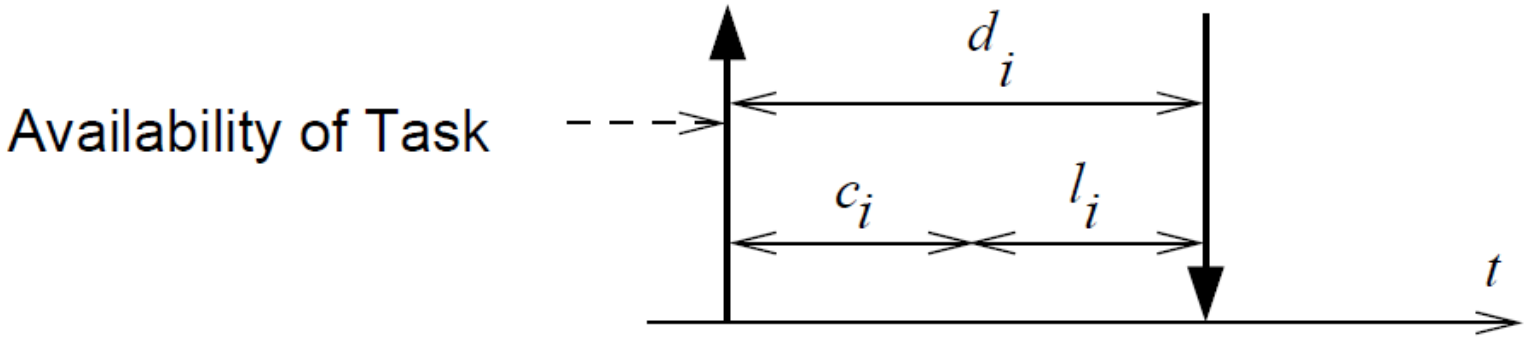
Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 12 月 12 日

Aperiodic scheduling:
- Scheduling with no precedence constraints -

Let $\{T_i\}$ be a set of tasks. Let:

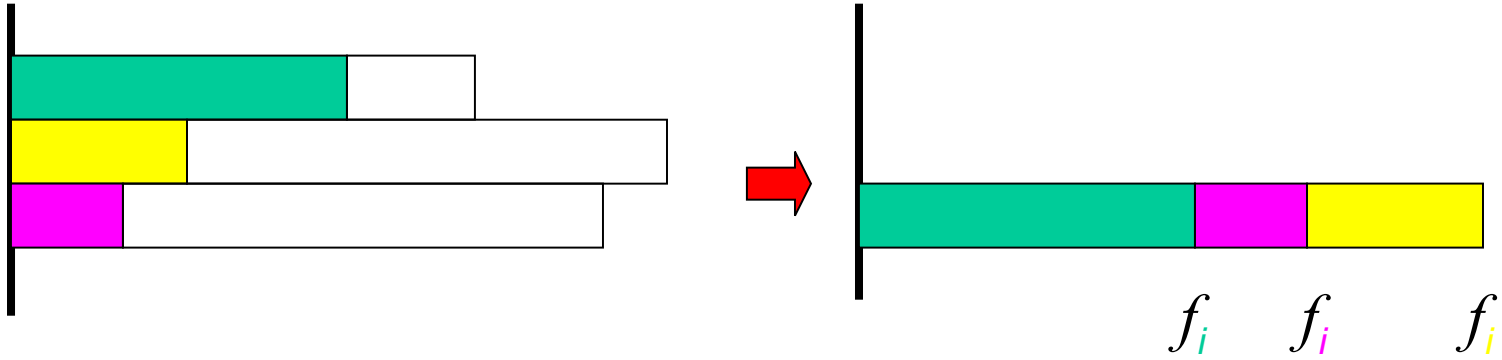
- c_i be the execution time of T_i,
- d_i be the **deadline interval**, that is,
  the time between T_i becoming available
  and the time until which T_i has to finish execution.
- l_i be the **laxity** or **slack**, defined as $l_i = d_i - c_i$
- f_i be the finishing time.



Uniprocessor with equal arrival times

Preemption is useless.

Earliest Due Date (EDD): Execute task with earliest due date (deadline) first.



EDD requires all tasks to be sorted by their (absolute) deadlines. Hence, its complexity is $O(n \log(n))$.

Optimality of EDD

EDD is optimal, since it follows Jackson's rule:
Given a set of n independent tasks, any algorithm that executes the tasks in order of non-decreasing (absolute) deadlines is optimal with respect to minimizing the maximum lateness.

Proof (See Buttazzo, 2002):

- Let τ be a schedule produced by any algorithm A
- If $A \neq$ EDD $\rightarrow \exists\ T_a, T_b, d_a \leq d_b, T_b$ immediately precedes T_a in τ.
- Let τ' be the schedule obtained by exchanging T_a *and* $T_{b.}$

Exchanging T_a and T_b cannot increase lateness

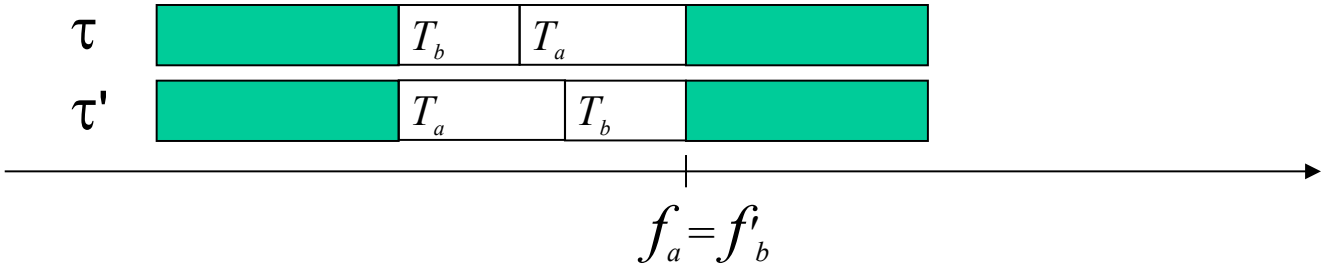
Max. lateness for T_a and T_b in τ is $L_{max}(a,b) = f_a - d_a$

Max. lateness for T_a and T_b in τ' is $L'_{max}(a,b) = max(L'_a, L'_b)$

Two possible cases

1. $L'_a \geq L'_b: \rightarrow L'_{max}(a,b) = f'_a - d_a < f_a - d_a = L_{max}(a,b)$
   since T_a starts earlier in schedule τ'.
2. $L'_a \leq L'_b: \rightarrow L'_{max}(a,b) = f'_b - d_b = f_a - d_b \leq f_a - d_a = L_{max}(a,b)$
   since $f_a = f'_b$ and $d_a \leq d_b$

☞ $L'_{max}(a,b) \leq L_{max}(a,b)$



EDD is optimal

- Any schedule τ with lateness L can be transformed into an EDD schedule τ^n with lateness $L^n \leq L$, which is the minimum lateness.
- EDD is optimal (q.e.d.)

Earliest Deadline First (EDF)
- Horn's Theorem -

Different arrival times: Preemption potentially reduces lateness.

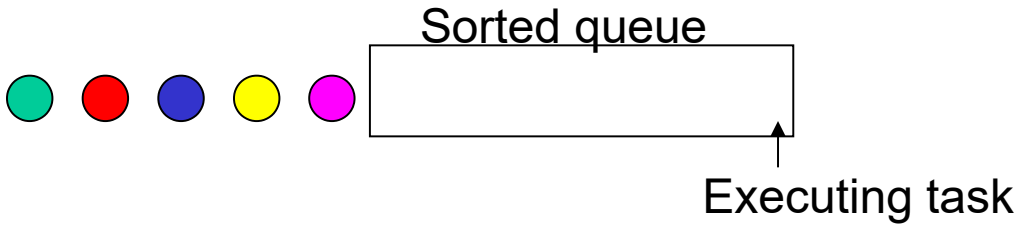
Theorem [Horn74]: Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among all the ready tasks is optimal with respect to minimizing the maximum lateness.

Earliest Deadline First (EDF)
- Algorithm -

Earliest deadline first (EDF) algorithm:

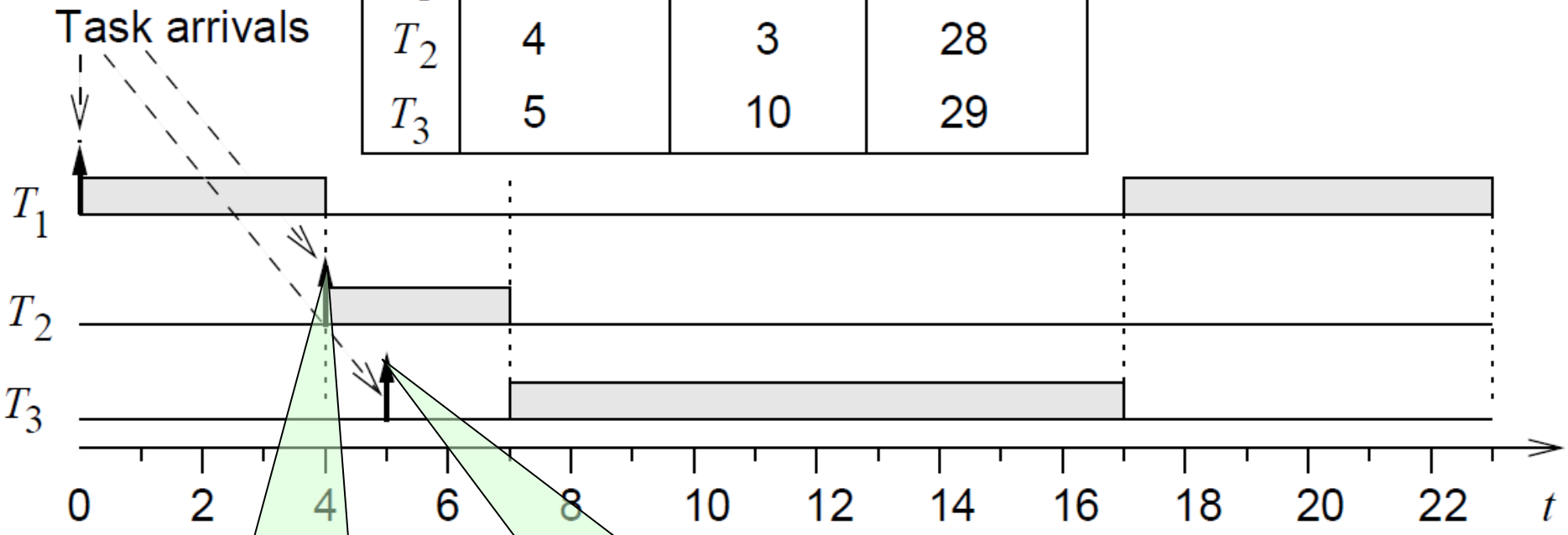
- Each time a new ready task arrives:
- It is inserted into a queue of ready tasks, sorted by their **absolute** deadlines. Task at head of queue is executed.
- If a newly arrived task is inserted at the head of the queue, the currently executing task is preempted.

Straightforward approach with sorted lists (full comparison with existing tasks for each arriving task) requires run-time $O(n^2)$; (less with binary search or bucket arrays).



Earliest Deadline First (EDF)
- Example -

	arrival	duration	deadline
T_1	0	10	33
T_2	4	3	28
T_3	5	10	29



Earlier deadline
☞ preemption

Later deadline
☞ no preemption

Optimality of EDF (1)

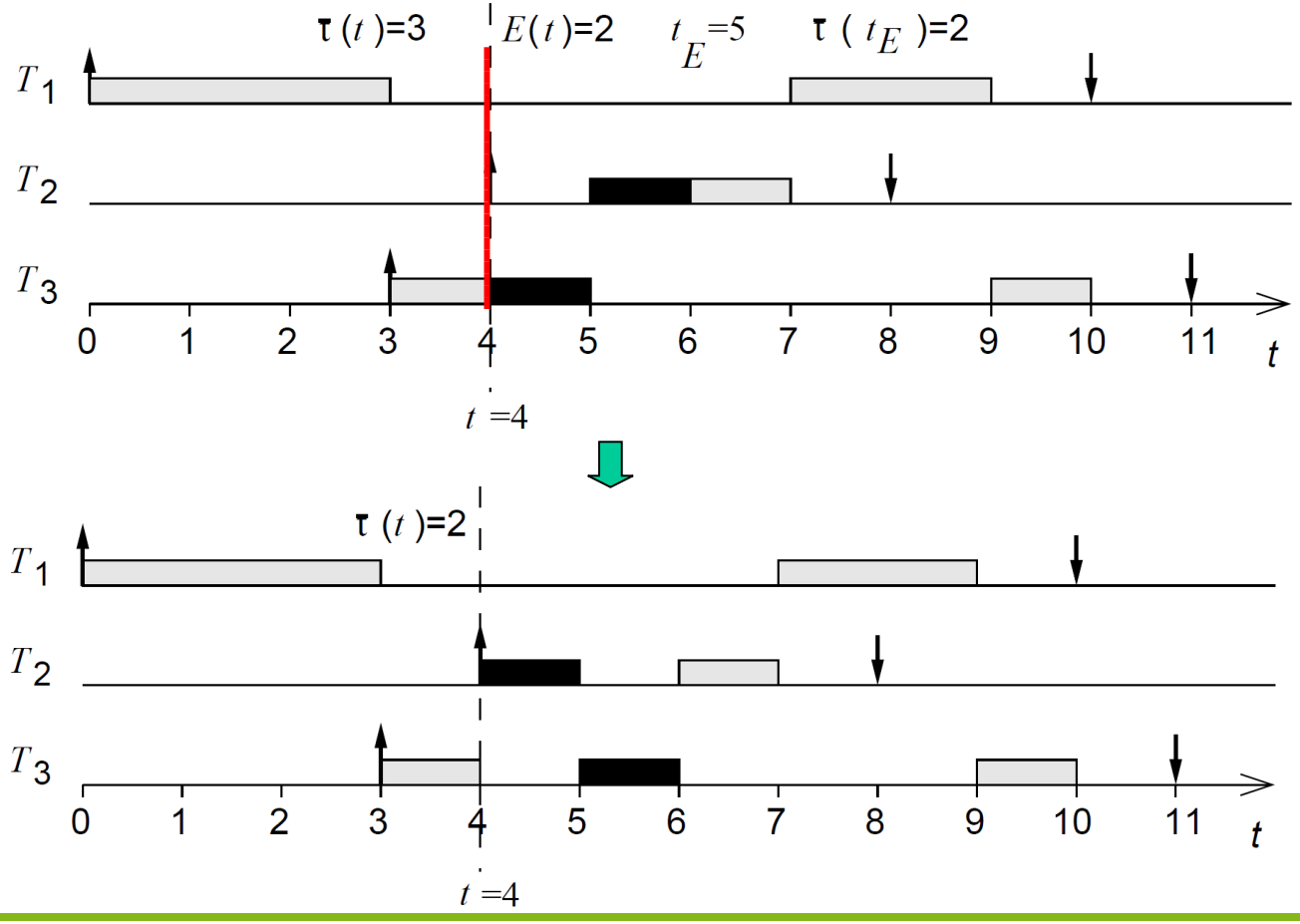
To be shown: EDF minimizes maximum lateness.
Proof (Buttazzo, 2002):

- Let τ be a schedule produced by generic schedule A
- Let τ_{EDF}: schedule produced by EDF
- Preemption allowed: tasks executed in disjoint time intervals
- τ divided into time slices of 1 time unit each
- Time slices denoted by $[t, t+1)$
- Let $\tau(t)$: task executing in $[t, t+1)$
- Let $E(t)$: task which, at time t, has the earliest deadline
- Let $t_E(t)$: time $(\geq t)$ at which the next slice of task $E(t)$ begins its execution in the current schedule

Optimality of EDF (2)

If $\tau \neq \tau_{EDF}$, then there exists time t: $\tau(t) \neq E(t)$

Idea: swapping $\tau(t)$ and $E(t)$ cannot increase max. lateness.



If $\tau(t)$ starts at t=0 and D=$\max_i\{d_i\}$ then τ_{EDF} can be obtained from τ by at most D transpositions

Optimality of EDF (3)

Algorithm **`interchange`**:
{ **for** (t=0 **to** D-1) {
if ($\tau(t) \neq E(t)$) {
$\tau(t_E) = \tau(t)$;
$\tau(t) = E(t)$; }}}

Using the same argument as in the proof of Jackson's algorithm, it is easy to show that swapping cannot increase maximum lateness; hence EDF is optimal.

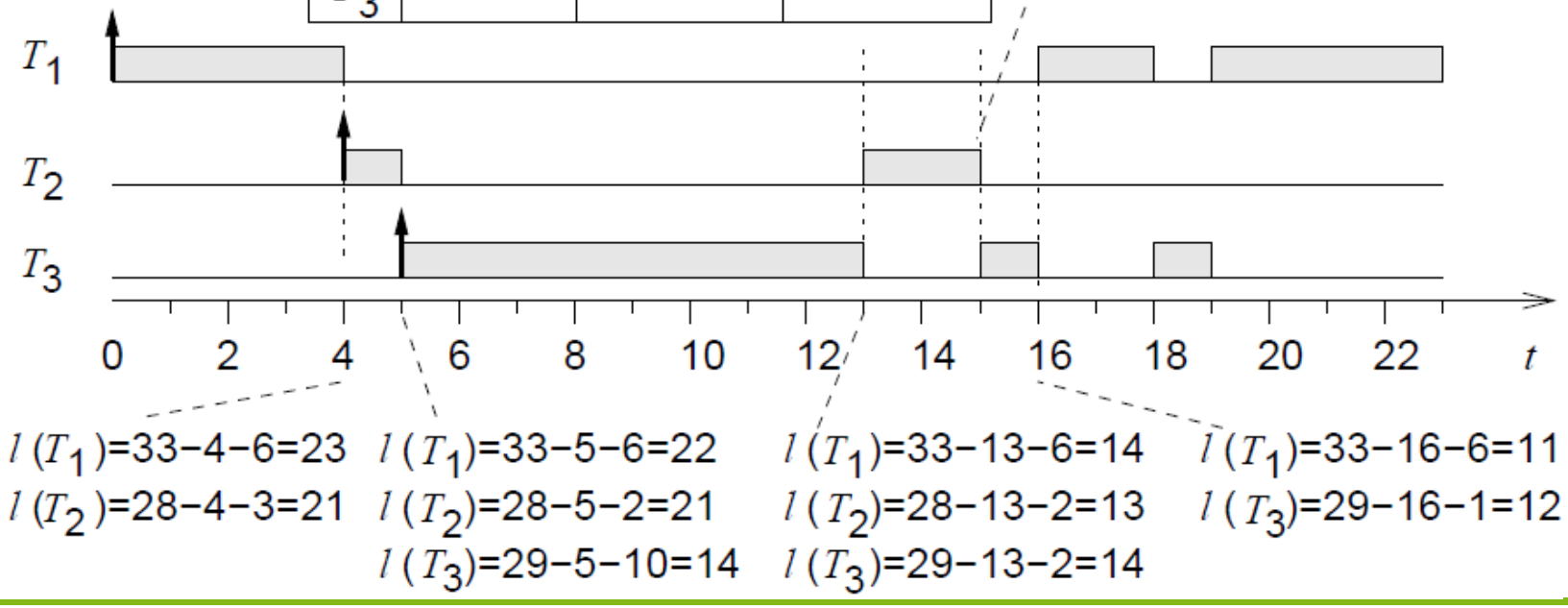
Does **`interchange`** preserve schedulability?

1. task $E(t)$ moved ahead: meeting deadline in new schedule if meeting deadline in τ
2. task $\tau(t)$ delayed: if $\tau(t)$ is feasible, then $(t_E+1) \leq d_E$, where d_E is the earliest deadline. Since $d_E \leq d_i$ for any i, we have $t_E+1 \leq d_i$, which guarantees schedulability of the delayed task. q.e.d.

Least laxity (LL), Least Slack Time First (LST)

Priorities = decreasing function of the laxity
(lower laxity ☞ higher priority); changing priority; preemptive.

	arrival	duration	deadline
T_1	0	10	33
T_2	4	3	28
T_3	5	10	29



Properties

- Not sufficient to call scheduler & re-compute laxity just at task arrival times.
- Overhead for calls of the scheduler.
- Many context switches.
- **Detects missed deadlines early.**
- LL is also an optimal scheduling for mono-processor systems.
- Dynamic priorities ☞ cannot be used with a fixed prio. OS.
- LL scheduling requires the knowledge of the execution time.

Scheduling without preemption (1)

Lemma: If preemption is not allowed, optimal schedules may have to leave the processor idle at certain times.

Proof: Suppose: optimal schedulers never leave processor idle.

Scheduling without preemption (2)

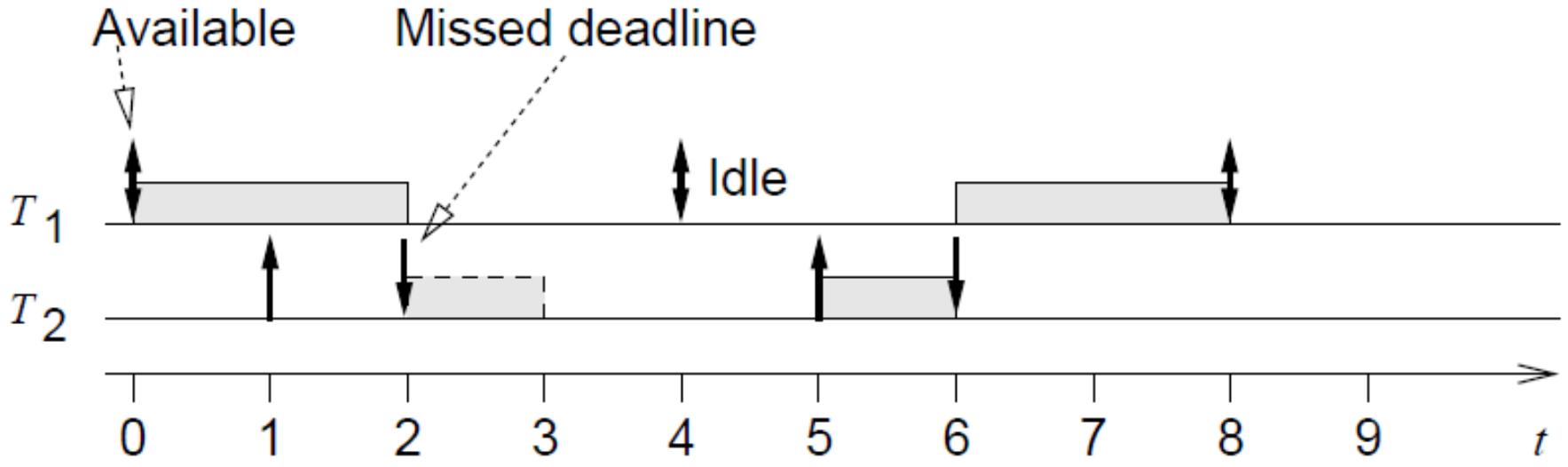
T_1: periodic, $c_1 = 2$, $p_1 = 4$, $d_1 = 4$

T_2: occasionally available at times $4*n+1$, $c_2 = 1$, $d_2 = 1$

T_1 has to start at $t = 0$

☞ deadline missed, but schedule is possible (start T_2 first)

☞ scheduler is not optimal ☞ contradiction! q.e.d.



Scheduling without preemption (3)

Preemption not allowed: ☞ optimal schedules may leave processor idle to finish tasks with early deadlines arriving late.

- ☞ Knowledge about the future is needed for optimal scheduling algorithms
- ☞ No online algorithm can decide whether or not to keep idle.

EDF is optimal among all scheduling algorithms not keeping the processor idle at certain times.

If arrival times are known a priori, the scheduling problem becomes NP-hard in general. B&B typically used.

= Branch and Bound

Summary

Definition mapping terms

- Resource allocation, assignment, binding, scheduling
- Hard vs. soft deadlines
- Static vs. dynamic (☞Time-triggered OS)
- Schedulability

Classical scheduling

- Aperiodic tasks
  - No precedences
    - Simultaneous (☞EDD)
    - Asynchronous Arrival Times (☞EDF, LL)

technische universität
dortmund

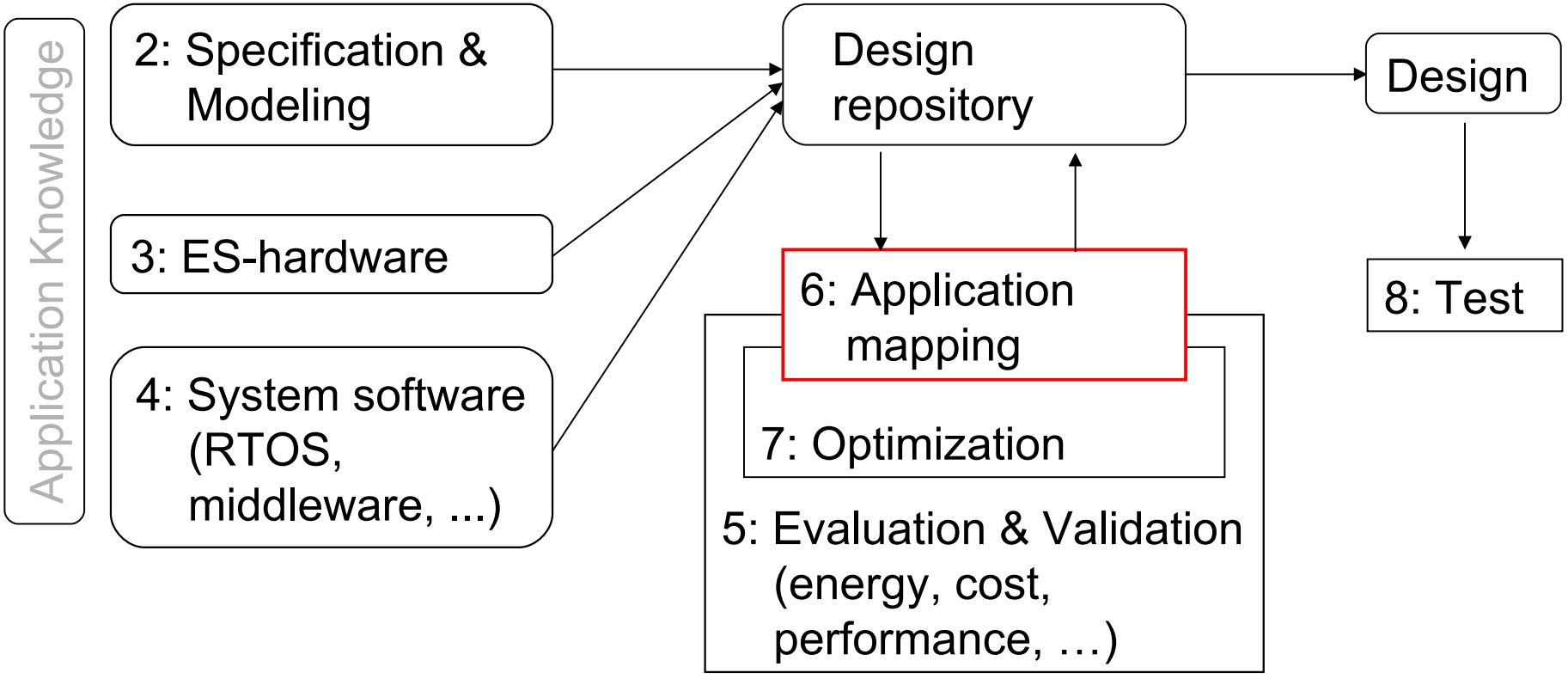
fakultät für informatik
informatik 12

Mapping of Applications to Platforms

Peter Marwedel
TU Dortmund,
Informatik 12

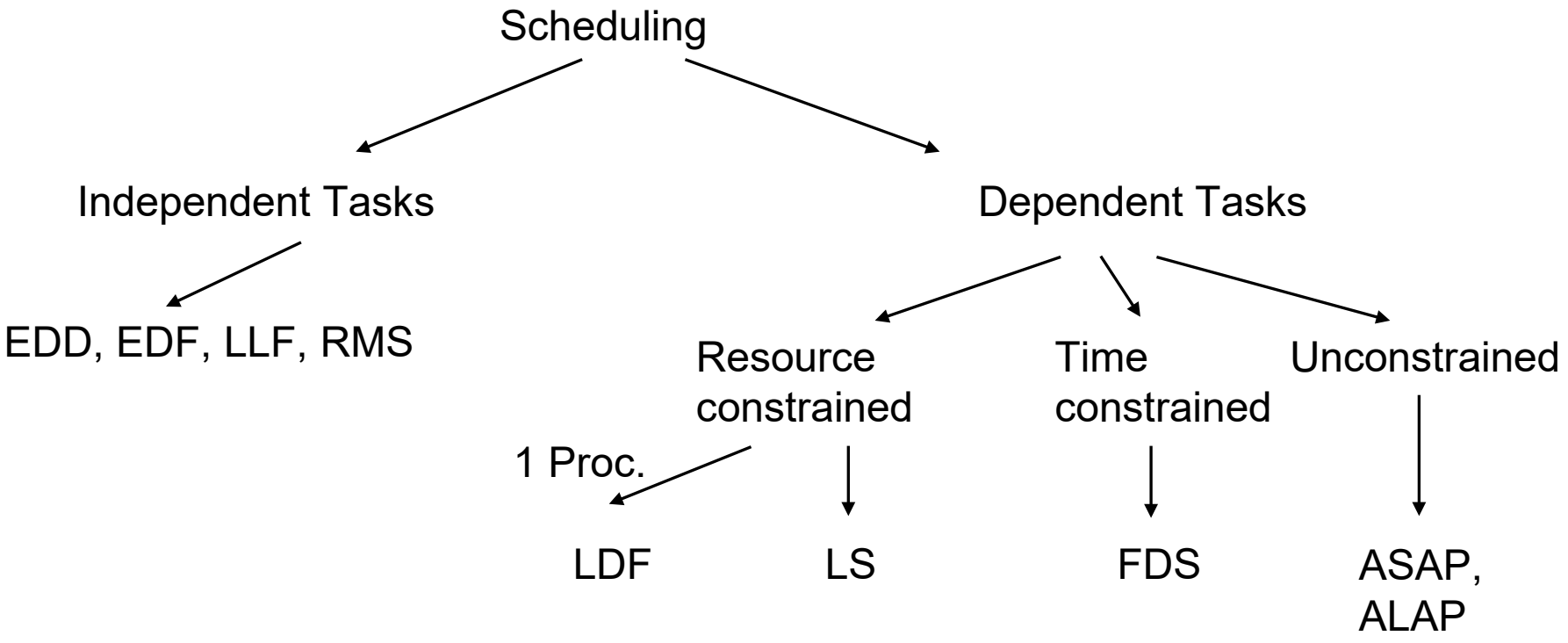
2012 年 12 月 18 日

Structure of this course



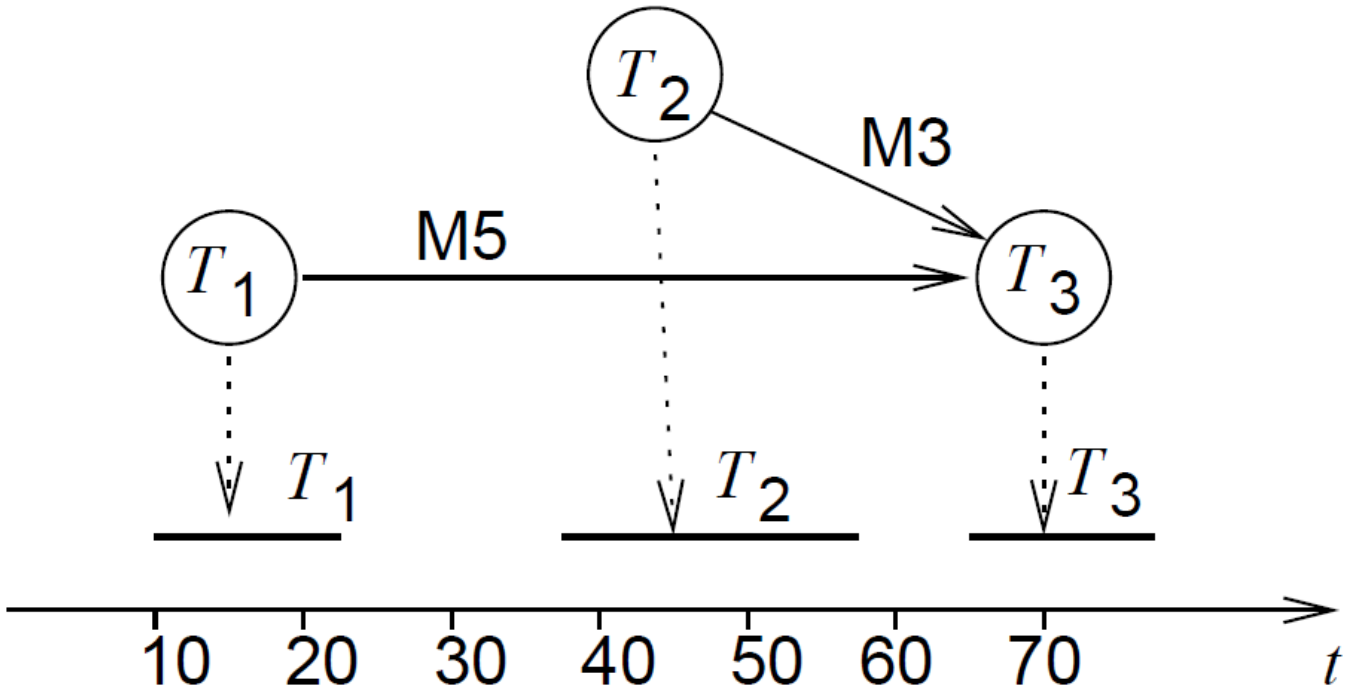
Numbers denote sequence of chapters

Classification of Scheduling Problems



Scheduling with precedence constraints

Task graph and possible schedule:

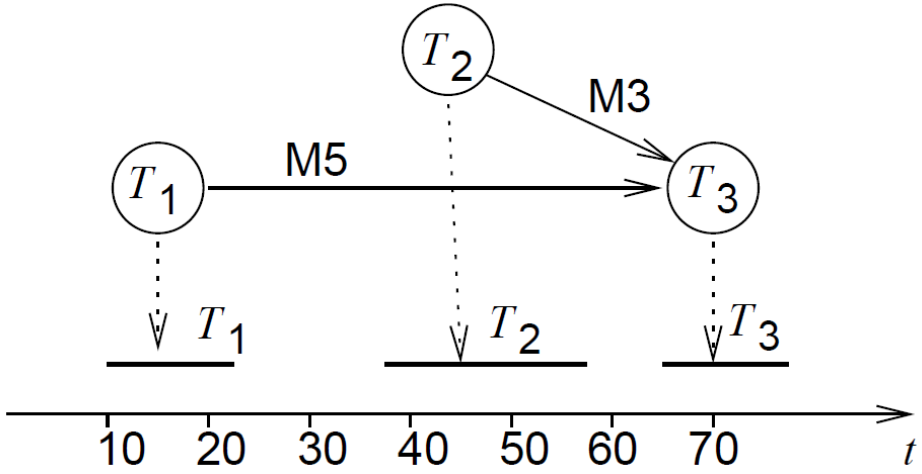


Simultaneous Arrival Times: The Latest Deadline First (LDF) Algorithm

LDF [Lawler, 1973]: reads the task graph and among the tasks with no successors inserts the one with the latest deadline into a queue. It then repeats this process, putting tasks whose successor have all been selected into the queue.

At run-time, the tasks are executed in the generated total order = opposite order of queuing.

LDF is non-preemptive and is optimal for mono-processors.



If no local deadlines exist, LDF performs just a topological sort.

Asynchronous Arrival Times: Modified EDF Algorithm

This case can be handled with a modified EDF algorithm.
The key idea is to transform the problem from a given set of dependent tasks into a set of independent tasks with different timing parameters [Chetto90].
This algorithm is optimal for mono-processor systems.

If preemption is not allowed, the heuristic algorithm developed by Stankovic and Ramamritham can be used.

Dependent tasks

The problem of deciding whether or not a schedule exists for a set of dependent tasks and a given deadline is NP-complete in general [Garey/Johnson].

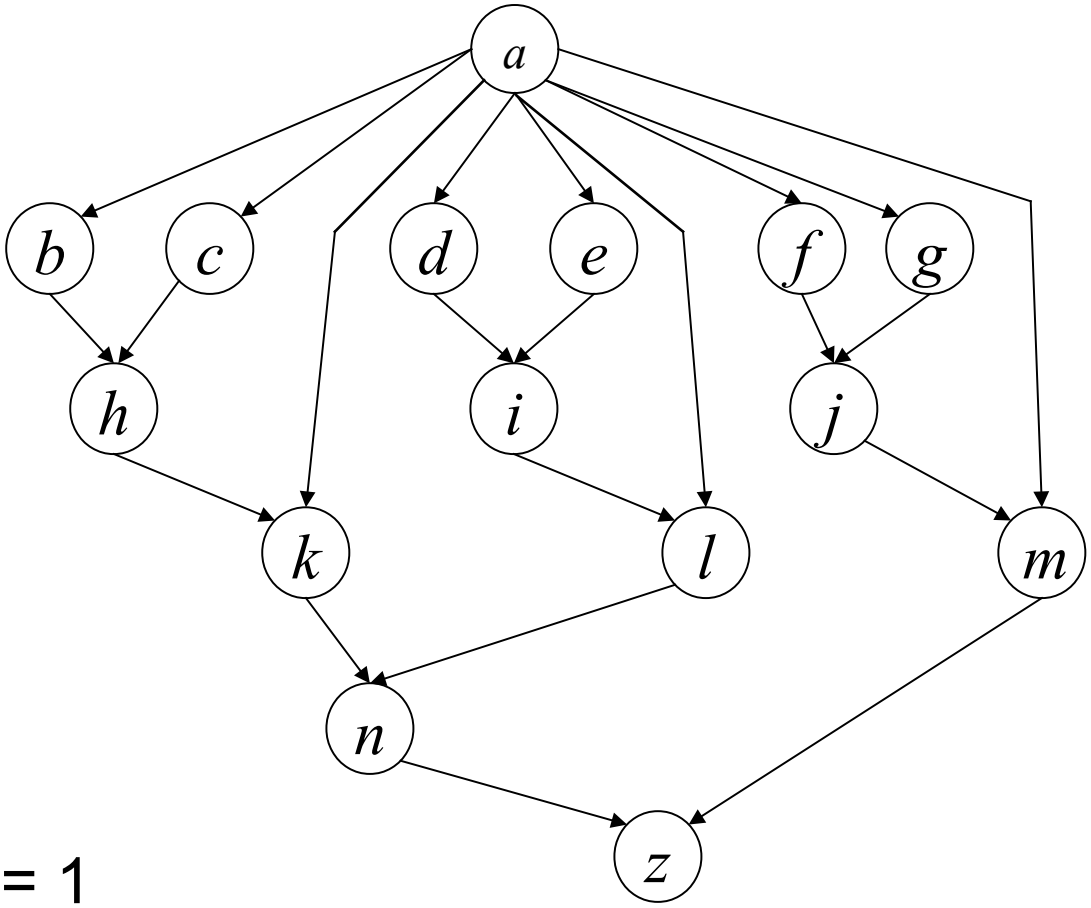
Strategies:

1. Add resources, so that scheduling becomes easier
2. Split problem into static and dynamic part so that only a minimum of decisions need to be taken at run-time.
3. Use scheduling algorithms from high-level synthesis

Classes of mapping algorithms considered in this course

- **Classical scheduling algorithms**
  Mostly for independent tasks & ignoring communication, mostly for mono- and homogeneous multiprocessors
- **Dependent tasks as considered in architectural synthesis**
  Initially designed in different context, but applicable
- **Hardware/software partitioning**
  Dependent tasks, heterogeneous systems, focus on resource assignment
- **Design space exploration using genetic algorithms**
  Heterogeneous systems, incl. communication modeling

Task graph



Assumption:
execution time = 1
for all tasks

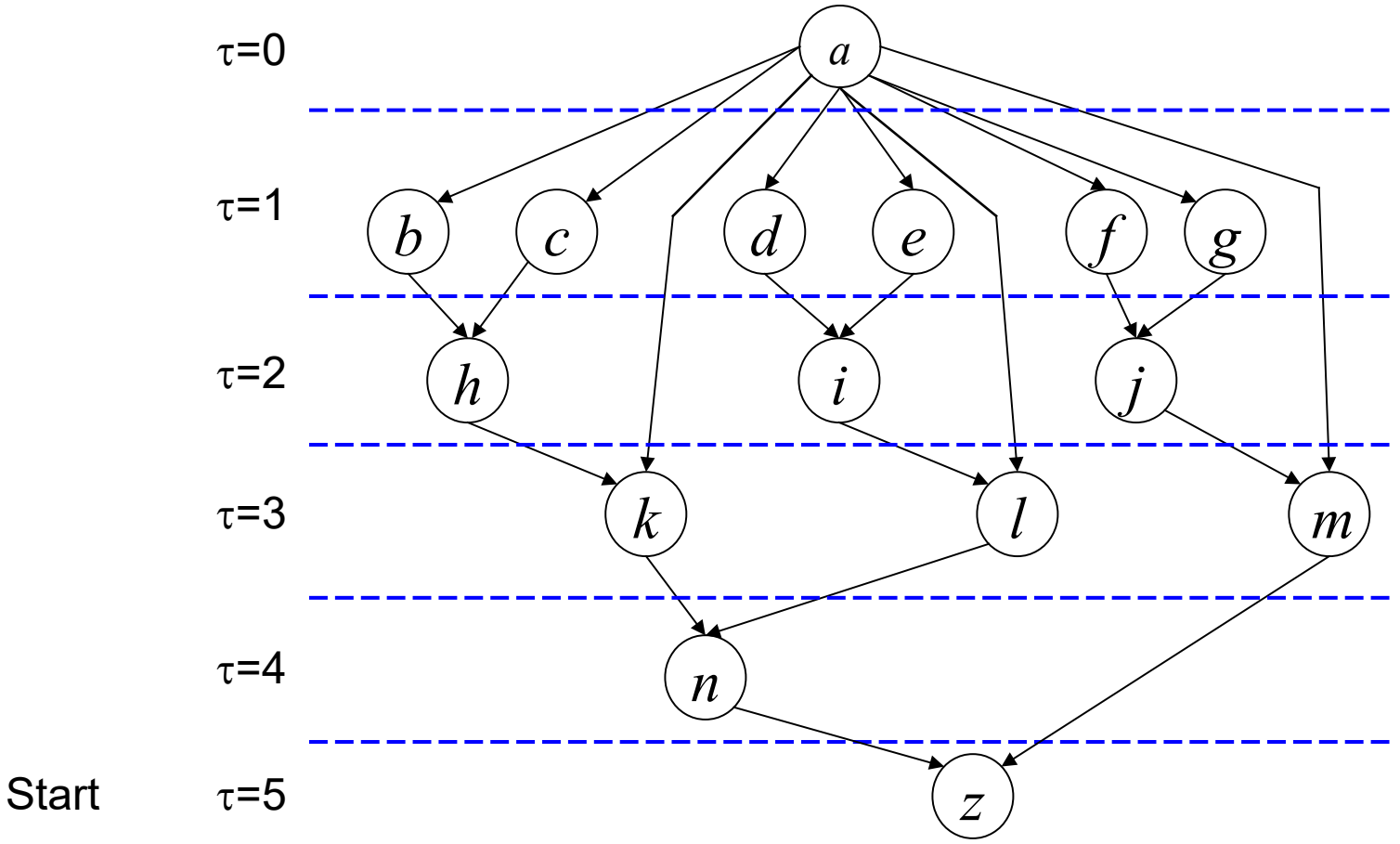
As-soon-as-possible (ASAP) scheduling

ASAP: All tasks are scheduled as early as possible

Loop over (integer) time steps:

- Compute the set of unscheduled tasks for which all predecessors have finished their computation
- Schedule these tasks to start at the current time step.

As-soon-as-possible (ASAP) scheduling: Example



As-late-as-possible (ALAP) scheduling

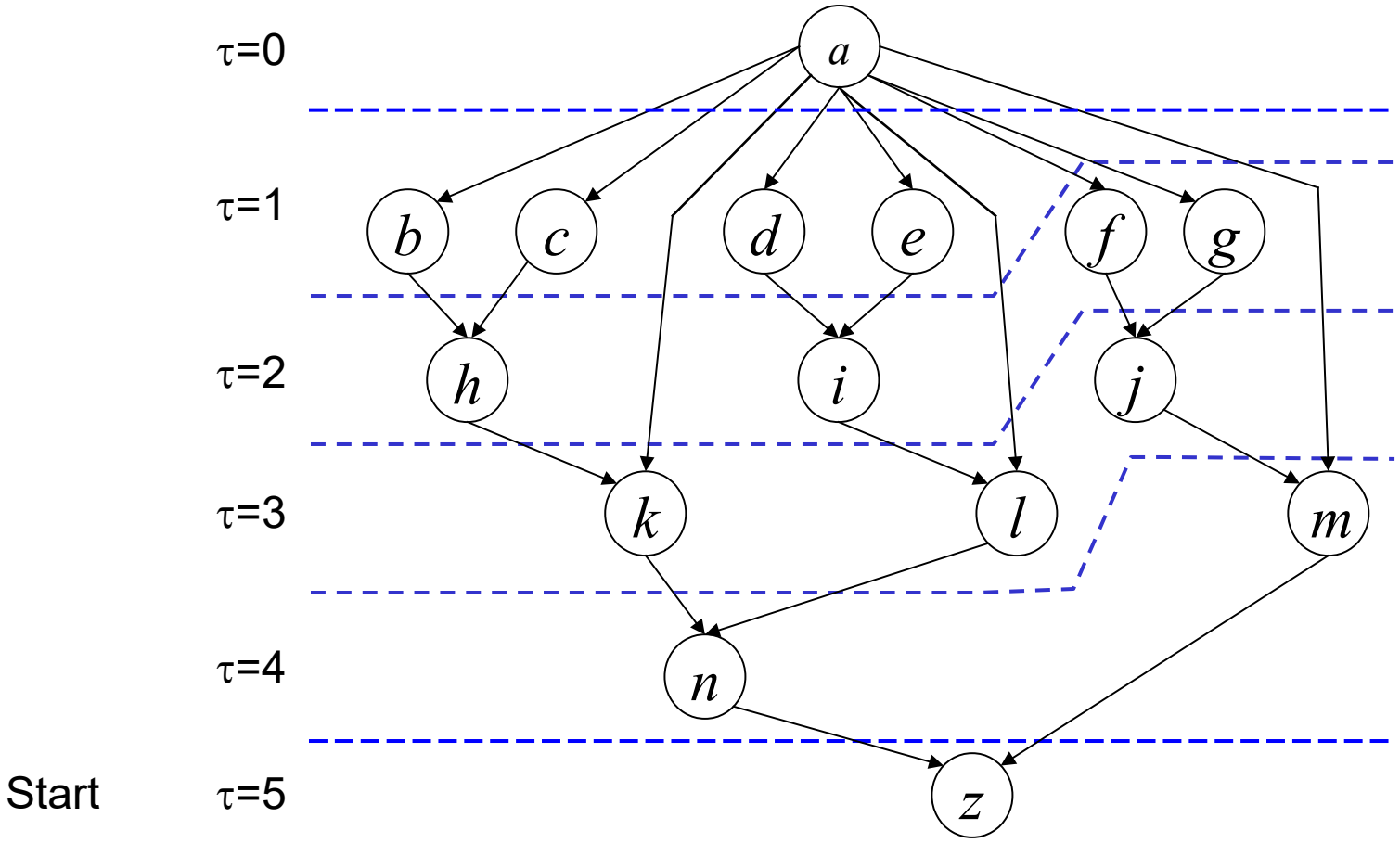
ALAP: All tasks are scheduled as late as possible

Start at last time step*:

- Schedule tasks with no successors and tasks for which all successors have already been scheduled.

* Generate a list, starting at its end

As-late-as-possible (ALAP) scheduling: Example



(Resource constrained) List Scheduling

List scheduling: extension of ALAP/ASAP method

Preparation:

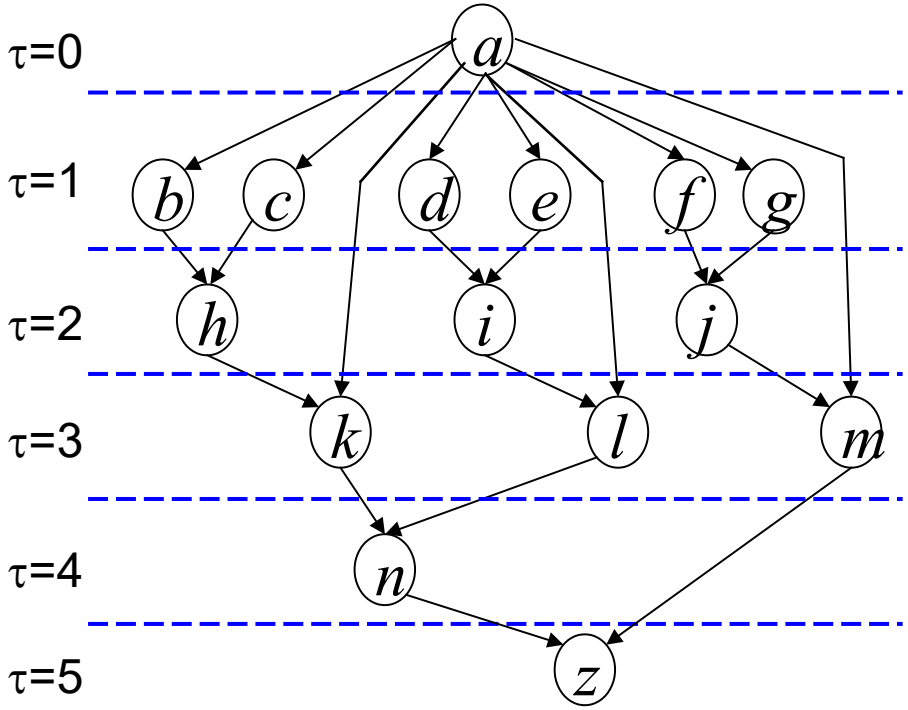
- Topological sort of task graph $G = (V,E)$
- Computation of priority of each task

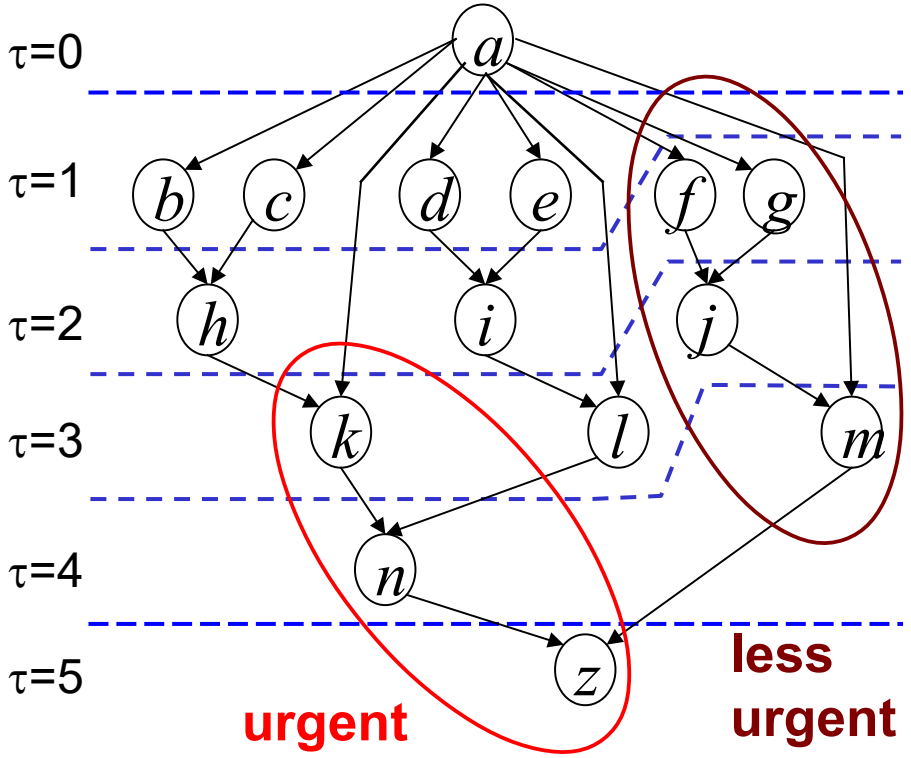
  Possible priorities u:

  - Number of successors
  - Longest path
  - **Mobility** = τ (ALAP schedule) - τ (ASAP schedule)

Mobility as a priority function

Mobility is not very precise





Algorithm

List($G(V,E)$, B, u){
i :=0;
repeat {
Compute set of candidate tasks A_i;
Compute set of not terminated tasks G_i;
Select $S_i \subseteq A_i$ of maximum priority r such that
$| S_i | + | G_i | \leq B$ (***resource constraint***)
foreach ($v_j \in S_i$): τ (v_j):=i; (*set start time*)
i := i +1;
}
until (all nodes are scheduled);
return (τ);
}

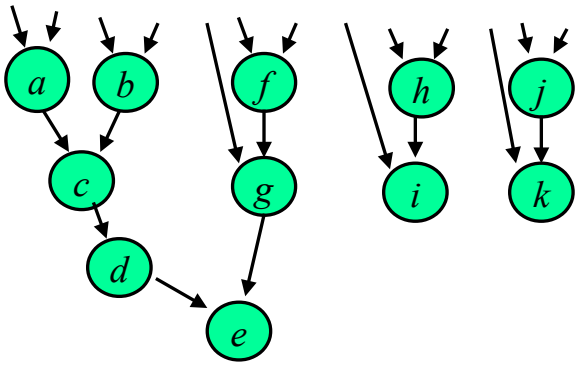
may be repeated for different task/ processor classes

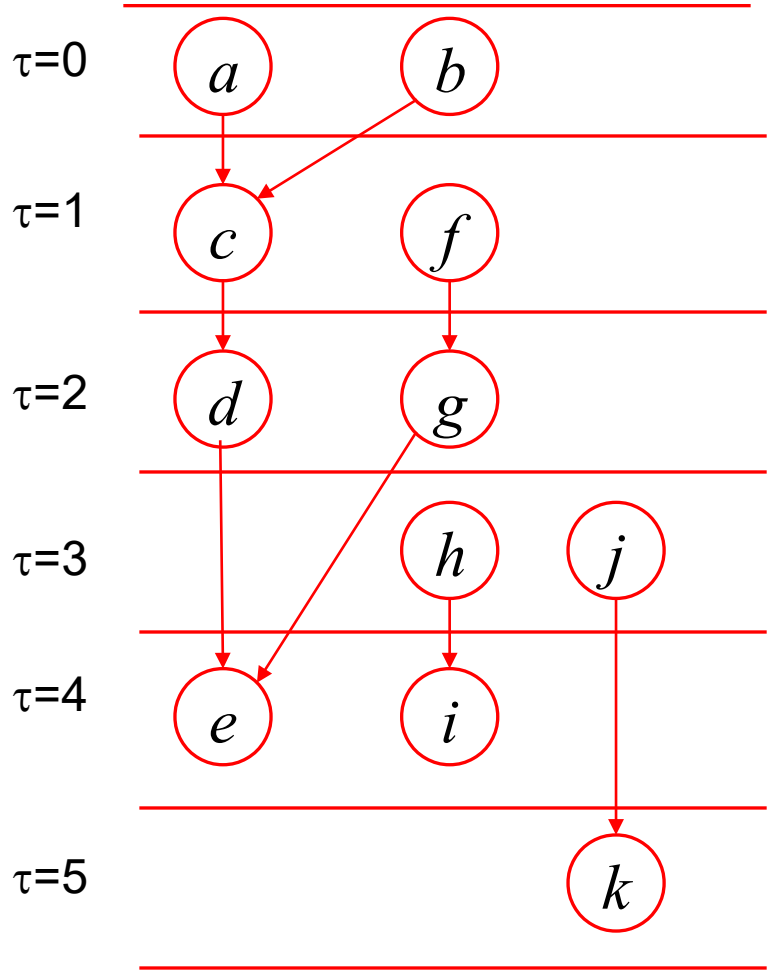
Complexity: $O(|V|)$

Example

Assuming $B = 2$, unit execution time and u : path length

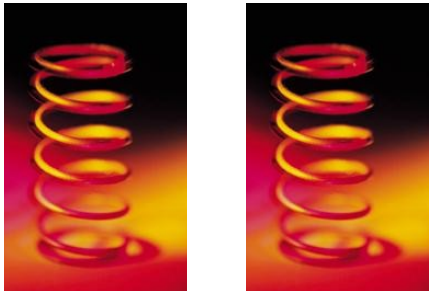
$$u(a) = u(b) = 4$$
$$u(c) = u(f) = 3$$
$$u(d) = u(g) = u(h) = u(j) = 2$$
$$u(e) = u(i) = u(k) = 1$$
$$\forall\ i : G_i = 0$$





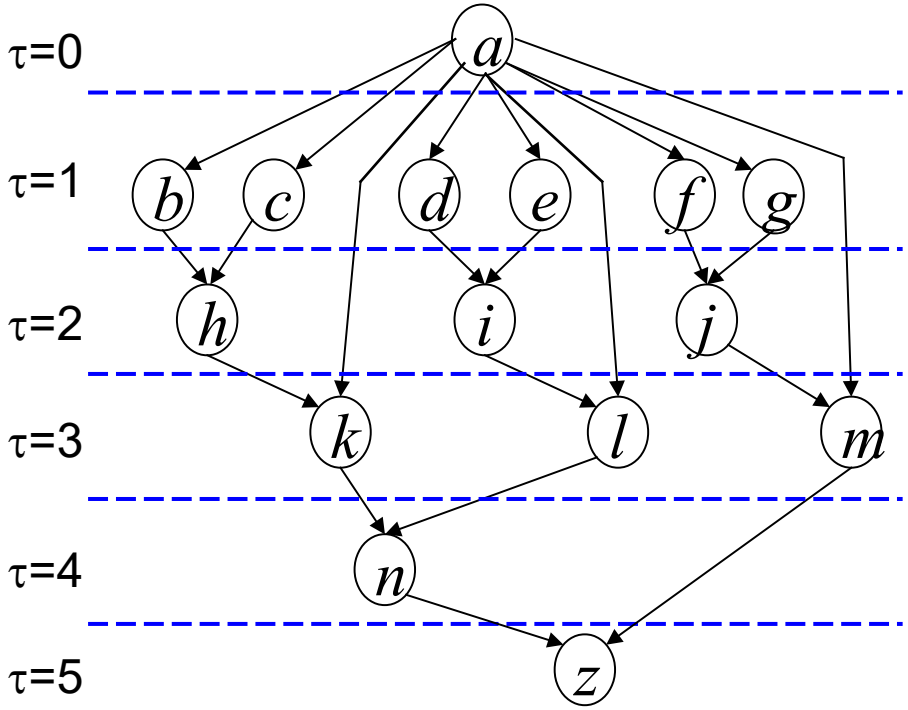
(Time constrained) Force-directed scheduling

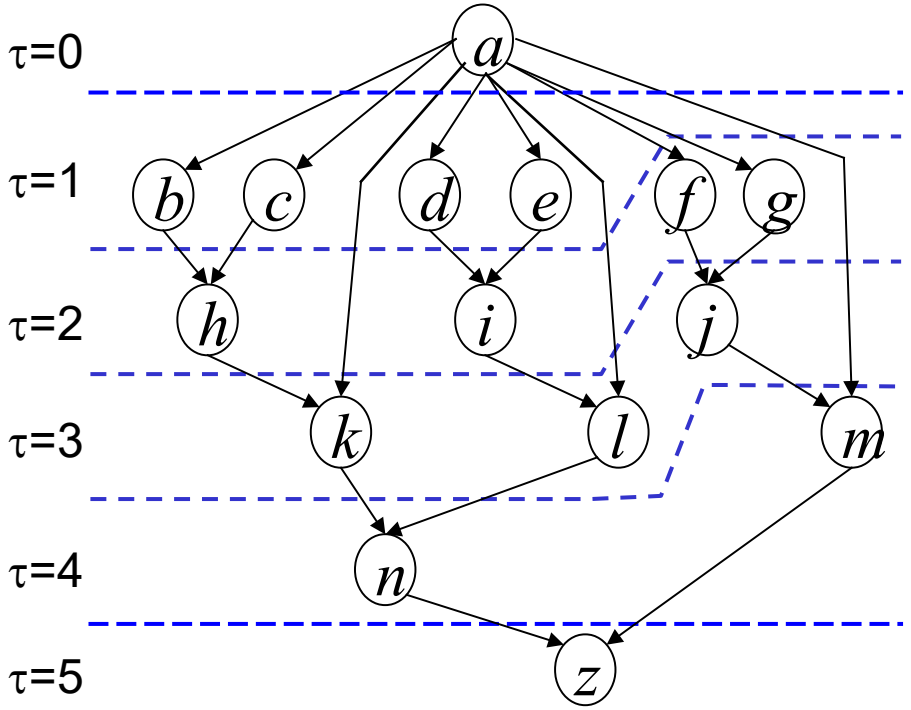
- Goal: balanced utilization of resources
- Based on spring model;
- Originally proposed for high-level synthesis



Pierre G. Paulin, J.P. Knight, Force-directed scheduling in automatic data path synthesis, *Design Automation Conference* (DAC), 1987, S. 195-202

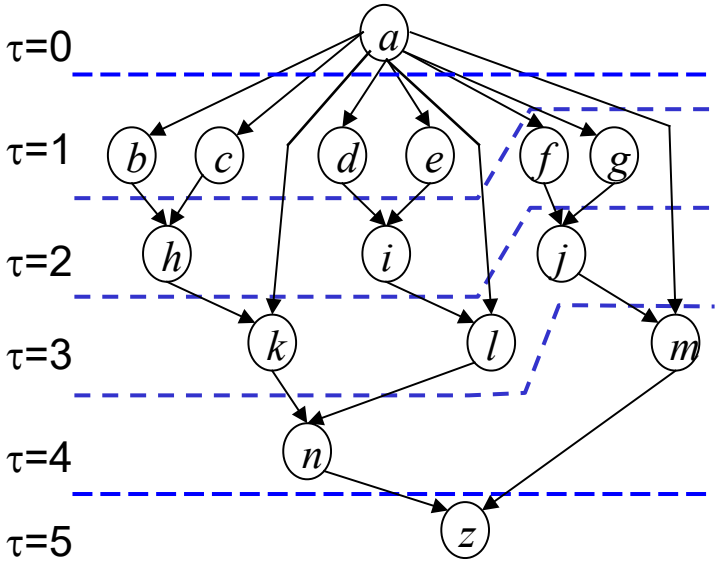
Phase 1: Generation of ASAP and ALAP Schedule

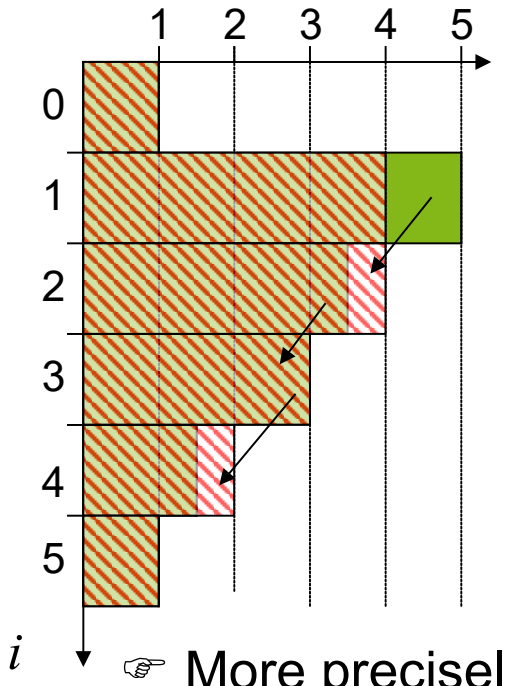




Next: computation of “forces”

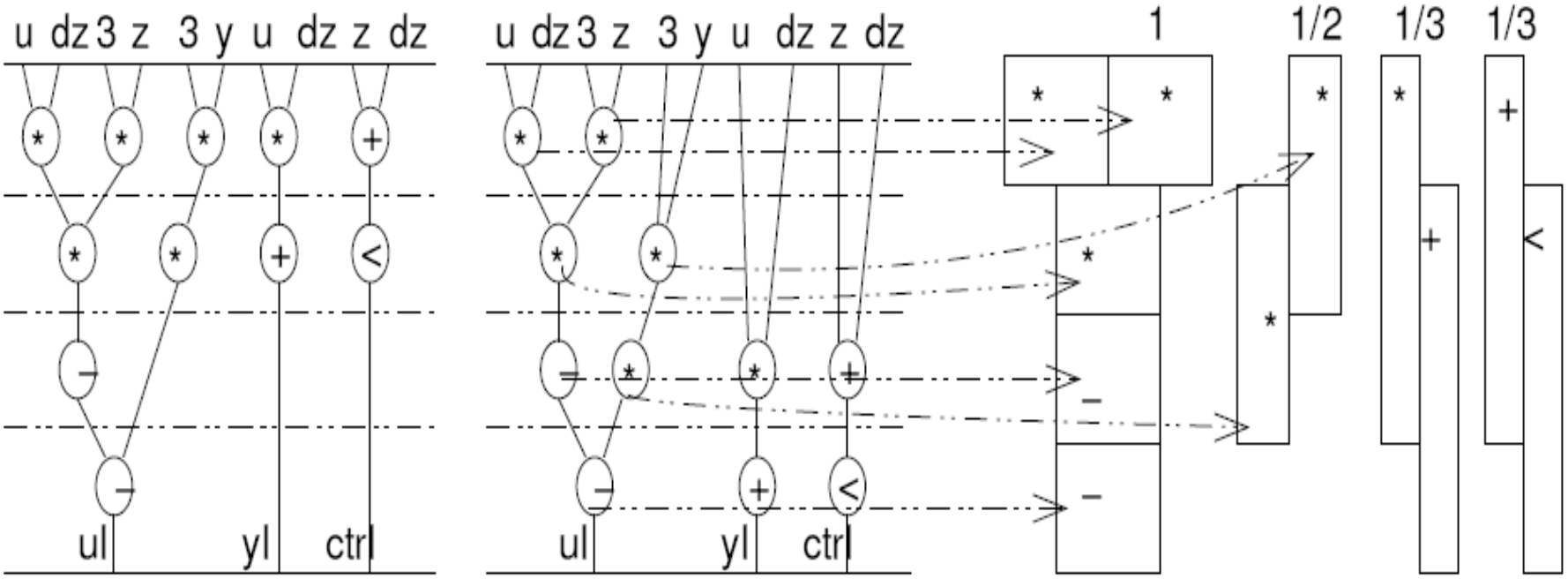
- Direct forces push each task into the direction of lower values of $D(i)$.
- Impact of direct forces on dependent tasks taken into account by indirect forces
- Balanced resource usage ≈ smallest forces
- For our simple example and time constraint = 6: result = ALAP schedule





☞ More precisely …

1. Compute time frames $R(j)$
2. Compute "probability" $P(j,i)$ of assignment $j \rightarrow i$

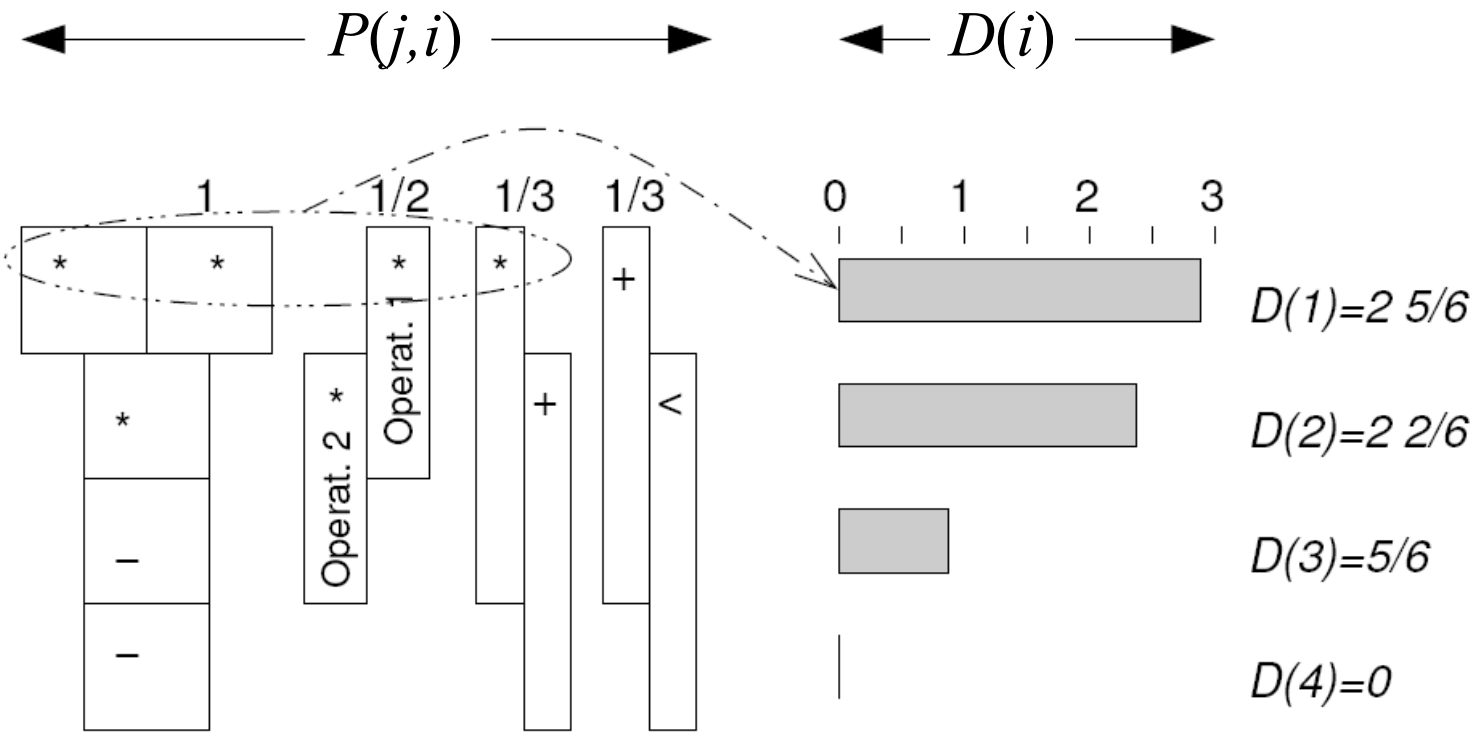


$R(j)$ = {ASAP-control step … ALAP-control step}

$$P(j,i) = \begin{cases} \frac{1}{|R(j)|} & \text{if} \quad i \in R(j) \\ 0 & \text{otherwise} \end{cases}$$

3. Compute "distribution" $D(i)$ (# Operations in control step i)

$$D(i) = \sum_{j, type(j) \in H} P(j,i)$$

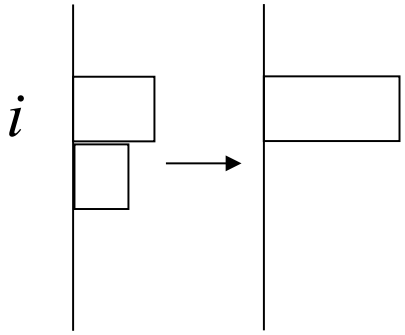


4. Compute direct forces (1)

- $\Delta P_i(j,i')$: Δ for force on task j in time step i', if j is mapped to time step i.

  The new probability for executing j in i is 1; the previous was $P(j,i)$.

  The new probability for executing j in $i' \neq i$ is 0; the previous was $P(j,i)$.



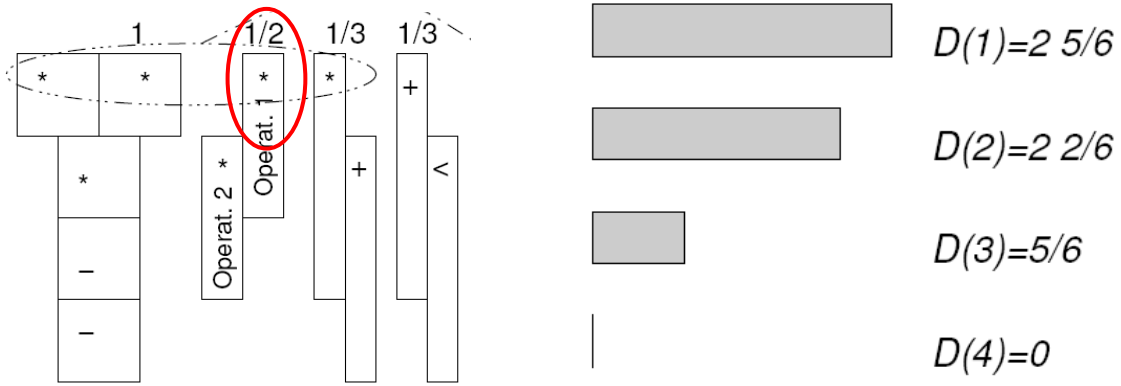
☞ $$\Delta P_i(j,i') = \begin{cases} 1 - P(j,i) & \text{if} \quad i = i' \\ -P(j,i') & \text{otherwise} \end{cases}$$

4. Compute direct forces (2)

- $SF(j, i)$ is the overall change of direct forces resulting from the mapping of j to time step i.

$$SF(j,i) = \sum_{i' \in R(j)} D(i') \Delta P_i(j,i') \qquad \Delta P_i(j,i') = \begin{cases} 1 - P(j,i) & \text{if} \quad i = i' \\ -P(j,i') & \text{otherwise} \end{cases}$$

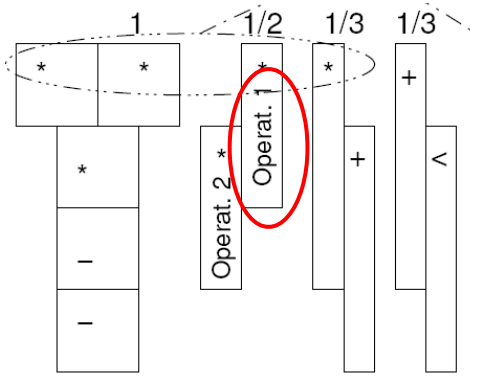
Example

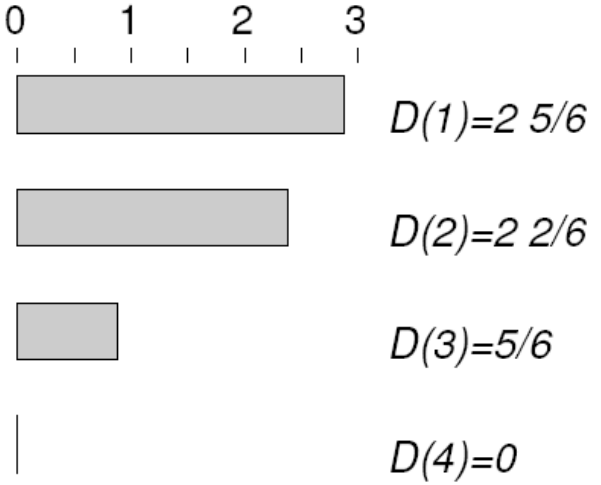


SF(1,1) = 2 5/6 · (1-1/2) – 2 2/6 · (1/2) = 1/2 · (17/6-14/6) = 1/4

4. Compute direct forces (3)

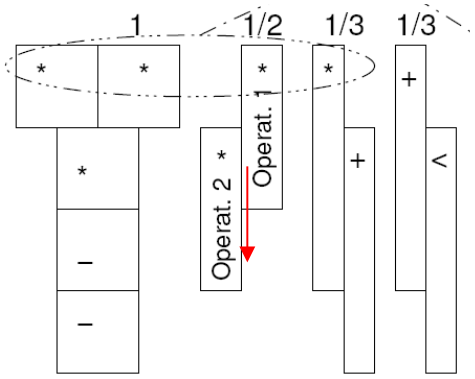
Direct force if task/operation 1 is mapped to time step 2





$$
\begin{aligned}
SF(1,2) &= D(1) * \Delta P_2(1,1) + D(2) * \Delta P_2(1,2) \\
&= 2\frac{5}{6} * (-0,5) + 2\frac{2}{6} * 0.5 \\
&= -\frac{17}{12} + \frac{14}{12} \\
&= -\frac{3}{12} = -\frac{1}{4}
\end{aligned}
$$

5. Compute indirect forces (1)



Mapping task 1 to time step 2 implies mapping task 2 to time step 3

☞ Consider predecessor and successor forces:

$$VF(j,i) = \sum_{j'} \sum_{j' \in \text{predecessor of } j} \sum_{I} D(i')\Delta P_{j,i}(j',i')$$

$$NF(j,i) = \sum_{j'} \sum_{j' \in \text{successor of } j} \sum_{I} D(i')\Delta P_{j,i}(j',i')$$

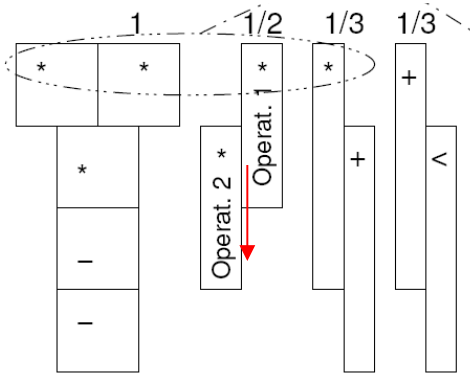
$\Delta P_{j,i}(j',i')$ is the Δ in the probability of mapping j' to i' resulting from the mapping of j to i

5. Compute indirect forces (2)

$$VF(j,i) = \sum_{j' \in \text{predecessor of } j} \sum_{i' \in I} D(i') \Delta P_{j,i}(j', i')$$

$$NF(j,i) = \sum_{j' \in \text{successor of } j} \sum_{i' \in I} D(i') \Delta P_{j,i}(j', i')$$

Example: Computation of successor forces for task 1 in time step 2



$$\begin{aligned} NF(1,2) &= D(2) * \Delta P_{1,2}(2,2) + D(3) * \Delta P_{1,2}(2,3) \\ &= 2\frac{2}{6} * (-0,5) + \frac{5}{6} * 0.5 \\ &= -\frac{14}{12} + \frac{5}{12} \\ &= -\frac{9}{12} = -\frac{3}{4} \end{aligned}$$

Overall forces

The total force is the sum of direct and indirect forces:

$$F(j,i) = SF(j,i) + VF(j,i) + NF(j,i)$$

In the example:

$$F(1,2) = SF(1,2) + NF(1,2) = -\frac{1}{4} + (-\frac{3}{4}) = -1$$

The low value suggests mapping task 1 to time step 2

Overall approach

```
procedure forceDirectedScheduling;
  begin
     AsapScheduling;
     AlapScheduling;
     while not all tasks scheduled do
        begin
           select task T with smallest total force;
           schedule task T at time step minimizing forces;
           recompute forces;
        end;
  end
```

May be repeated for different task/ processor classes

Not sufficient for today's complex, heterogeneous hardware platforms

Evaluation of HLS-Scheduling

- Focus on considering dependencies
- Mostly heuristics, few proofs on optimality
- Not using global knowledge about periods etc.
- Considering discrete time intervals
- Variable execution time available only as an extension
- Includes modeling of heterogeneous systems

Overview

Scheduling of aperiodic tasks with real-time constraints:
Table with some known algorithms:

	Equal arrival times; non-preemptive	Arbitrary arrival times; preemptive
Independent tasks	EDD (Jackson)	EDF (Horn)
Dependent tasks	LDF (Lawler), ASAP, ALAP, LS, FDS	EDF* (Chetto)

Conclusion

- HLS-based scheduling
  - ASAP
  - ALAP
  - *List scheduling* (LS)
  - *Force-directed scheduling* (FDS)
- Evaluation

technische universität
dortmund

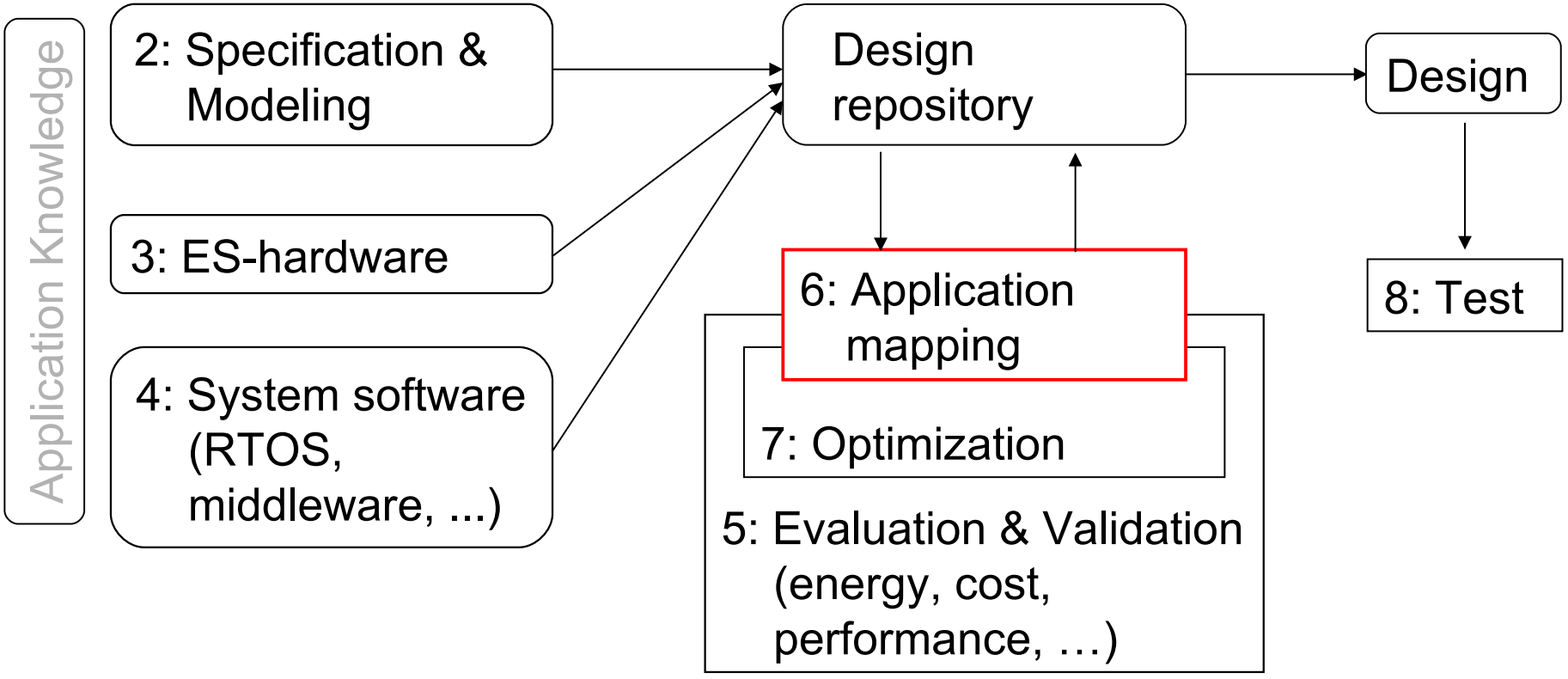
fakultät für informatik
informatik 12

Classical scheduling algorithms for periodic systems

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 12 月 19 日

Structure of this course

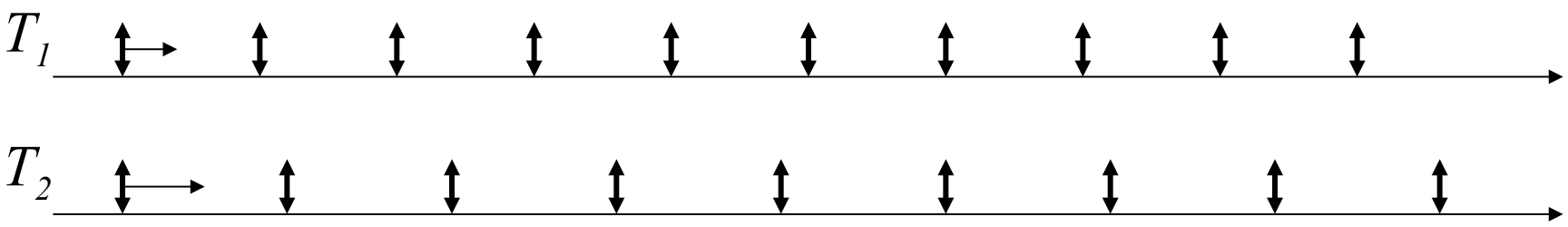


Numbers denote sequence of chapters

Classes of mapping algorithms considered in this course

- **Classical scheduling algorithms**
  Mostly for independent tasks & ignoring communication, mostly for mono- and homogeneous multiprocessors
- **Dependent tasks as considered in architectural synthesis**
  Initially designed in different context, but applicable
- **Hardware/software partitioning**
  Dependent tasks, heterogeneous systems, focus on resource assignment
- **Design space exploration using genetic algorithms;**
  Heterogeneous systems, incl. communication modeling

Periodic scheduling



Each execution instance of a task is called a **job**.

Notion of optimality for aperiodic scheduling does not make sense for periodic scheduling.

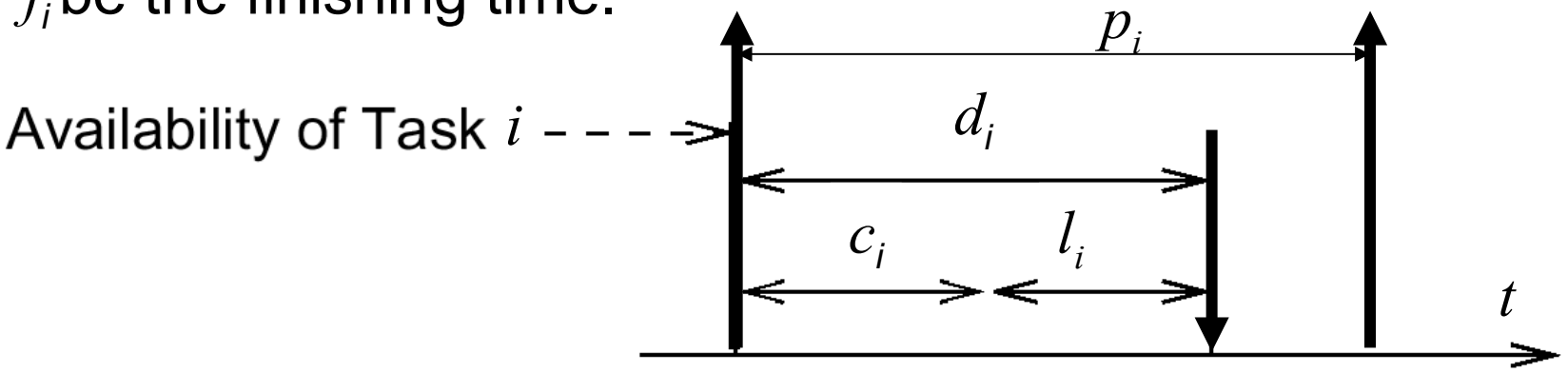
For periodic scheduling, the best that we can do is to design an algorithm which will always find a schedule if one exists.

☞ A scheduler is defined to be **optimal** iff it will find a schedule if one exists.

Periodic scheduling: Scheduling with no precedence constraints

Let $\{T_i\}$ be a set of tasks. Let:

- p_i be the period of task T_i,
- c_i be the execution time of T_i,
- d_i be the **deadline interval**, that is,
  the time between T_i becoming available
  and the time until which T_i has to finish execution.
- l_i be the **laxity** or **slack**, defined as $l_i = d_i - c_i$
- f_i be the finishing time.



Average utilization: important characterization of scheduling problems

Average utilization:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i}$$

Necessary condition for schedulability (with m=number of processors):

$$\mu \leq m$$

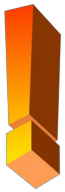
Independent tasks: Rate monotonic (RM) scheduling

Most well-known technique for scheduling independent periodic tasks [Liu, 1973].

Assumptions:

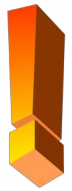
- All tasks that have hard deadlines are periodic.
- All tasks are independent.
- $d_i = p_i$, for all tasks.
- c_i is constant and is known for all tasks.
- The time required for context switching is negligible.
- For a single processor and for n tasks, the following equation holds for the average utilization μ:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n\left(2^{1/n} - 1\right)$$



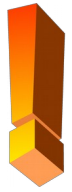
Rate monotonic (RM) scheduling
- The policy -

RM policy: **The priority of a task is a monotonically decreasing function of its period.**



At any time, a highest priority task among all those that are ready for execution is allocated.

Theorem: If all RM assumptions are met, schedulability is guaranteed.

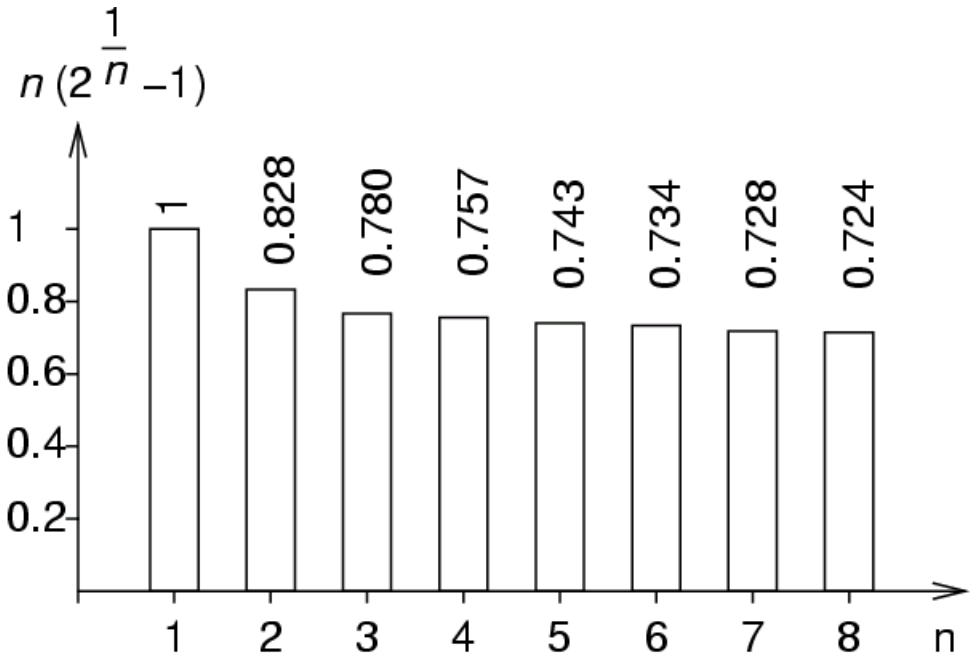


Maximum utilization for guaranteed schedulability

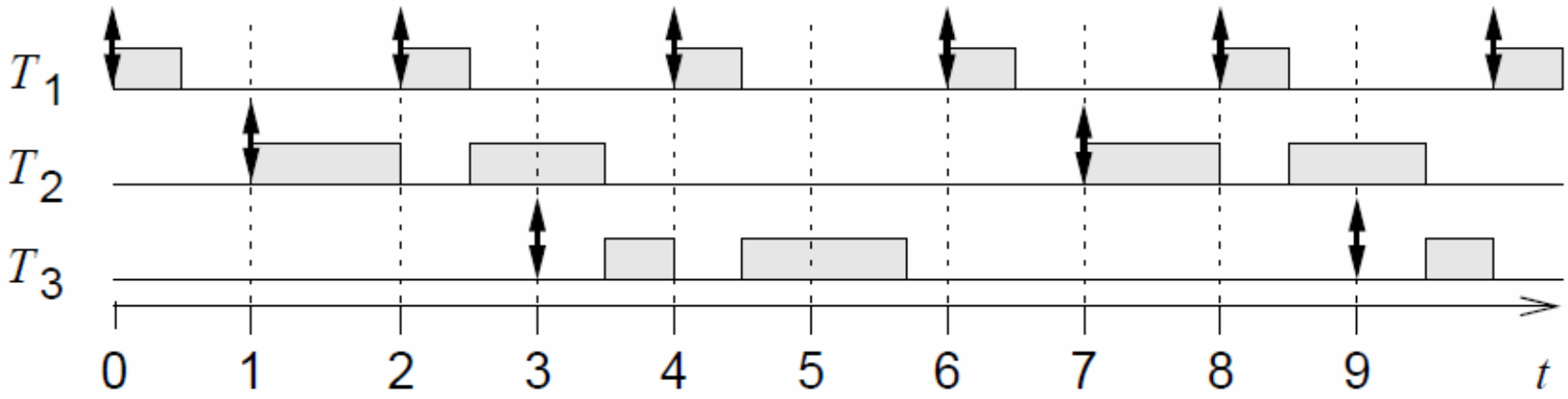
Maximum utilization as a function of the number of tasks:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n\left(2^{1/n} - 1\right)$$

$$\lim_{n \to \infty} \left(n\left(2^{1/n} - 1\right)\right) = \ln(2)$$



Example of RM-generated schedule



T_1 preempts T_2 and T_3.

T_2 and T_3 do not preempt each other.

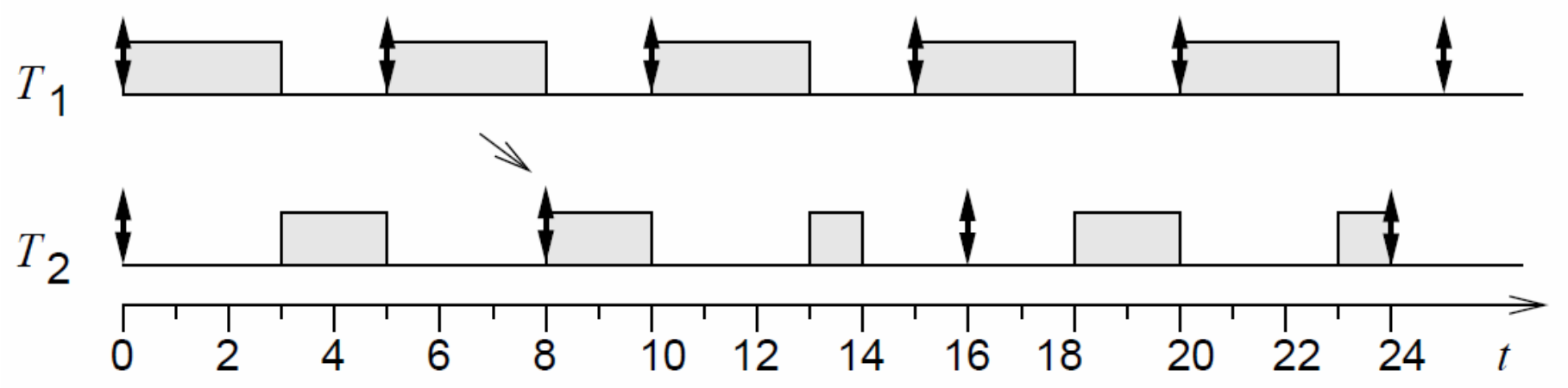
Failing RMS

Task 1: period 5, execution time 3

Task 2: period 8, execution time 3

$\mu = 3/5+3/8 = 24/40+15/40 = 39/40 \approx 0.975$

$2(2^{1/2} - 1) \approx 0.828$



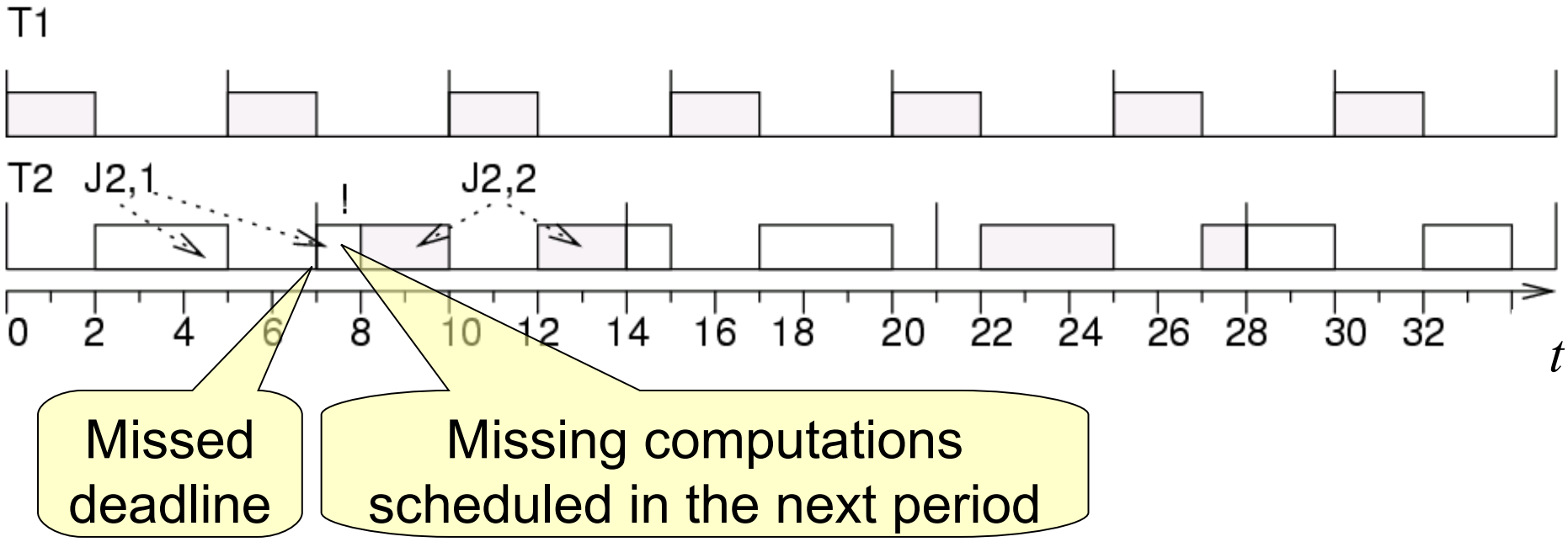
Case of failing RM scheduling

Task 1: period 5, execution time 2

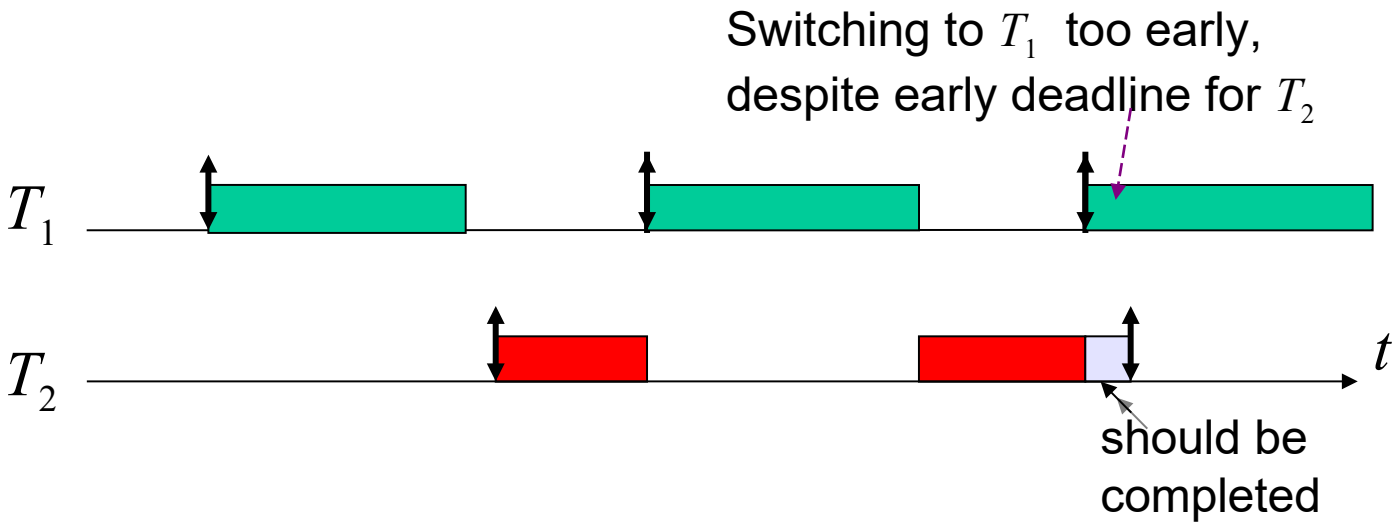
Task 2: period 7, execution time 4

$\mu = 2/5+4/7 = 34/35 \approx 0.97$

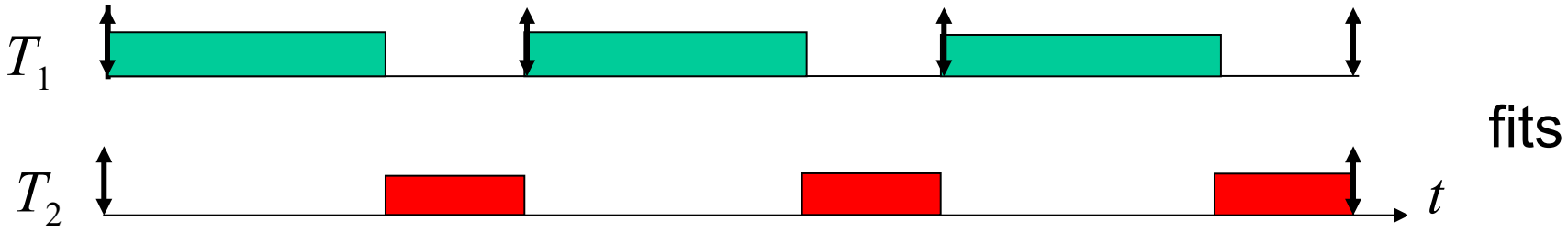
$2(2^{1/2} - 1) \approx 0.828$



Intuitively: Why does RM fail ?



No problem if $p_2 = m\, p_1, \quad m \in \mathbb{N}$:



Critical instants (1)

Definition

A **critical instant** of a task is the time at which the release of a task will produce the largest response time.

Lemma

For any task, the **critical instant** occurs if that task is simultaneously released with all higher priority tasks.

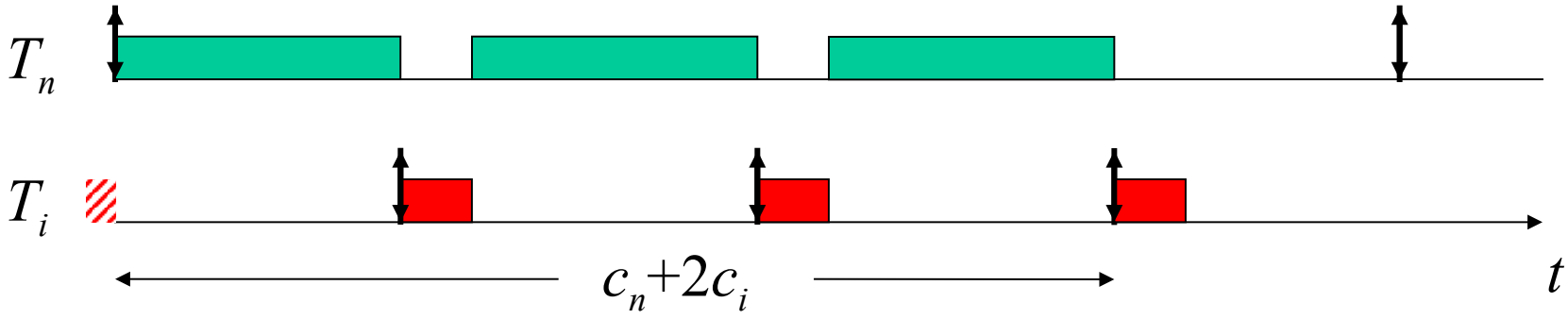
Proof

Let $T=\{T_1, \ldots, T_n\}$: periodic tasks with $\forall i: p_i \leq p_{i+1}$.

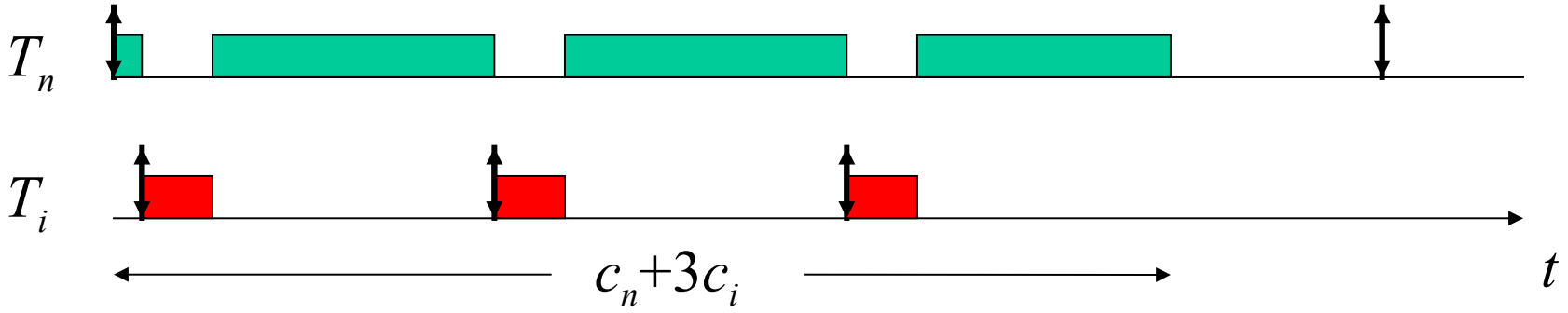
Source: G. Buttazzo, Hard Real-time Computing Systems, Kluwer, 2002

Critical instants (2)

Response time of T_n is delayed by tasks T_i of higher priority:



Delay may increase if T_i starts earlier



Maximum delay achieved if T_n and T_i start simultaneously.

Critical instants (3)

Repeating the argument for all $i = 1, \ldots n\text{-}1$:

- The worst case response time of a task occurs when it is released simultaneously with all higher-priority tasks. q.e.d.

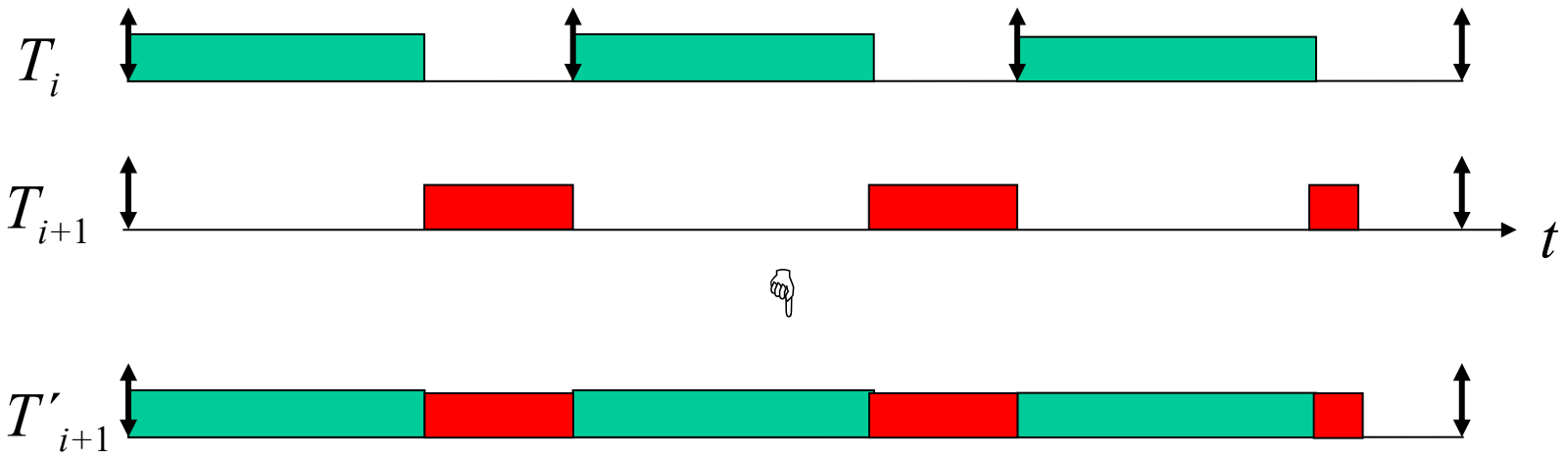
- Schedulability is checked at the critical instants.
- If all tasks of a task set are schedulable at their critical instants, they are schedulable at all release times.
- Observation helps designing examples

The case $\forall i: p_{i+1} = m_i p_i$

Lemma*: **If each task period is a multiple of the period of the next higher priority task**, then schedulability is also guaranteed if $\mu \leq 1$.

Proof: Assume schedule of T_i is given. Incorporate T_{i+1}:

T_{i+1} fills idle times of T_i; T_{i+1} completes in time, if $\mu \leq 1$.



Used as the higher priority task at the next iteration.

Properties of RM scheduling

- RM scheduling is based on **static** priorities. This allows RM scheduling to be used in an OS with static priorities, such as Windows NT.
- No idle capacity is needed if $\forall i: p_{i+1} = F\, p_i$:
  i.e. if the **period of each task is a multiple of the period of the next higher priority task**, schedulability is then also guaranteed if $\mu \leq 1$.
- A huge number of variations of RM scheduling exists.
- In the context of RM scheduling, many formal proofs exist.

EDF

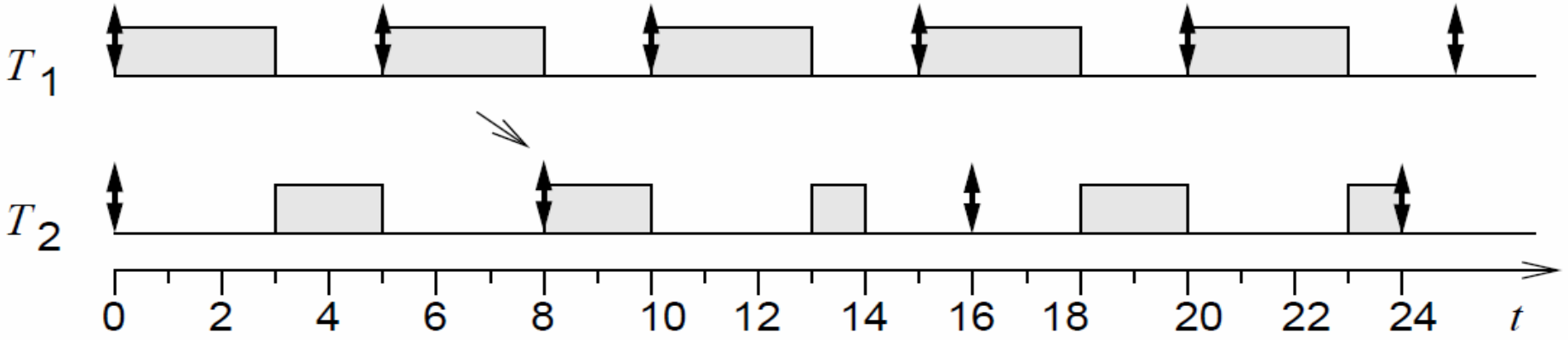
EDF can also be applied to periodic scheduling.

EDF optimal for every **hyper-period**
(= least common multiple of all periods)

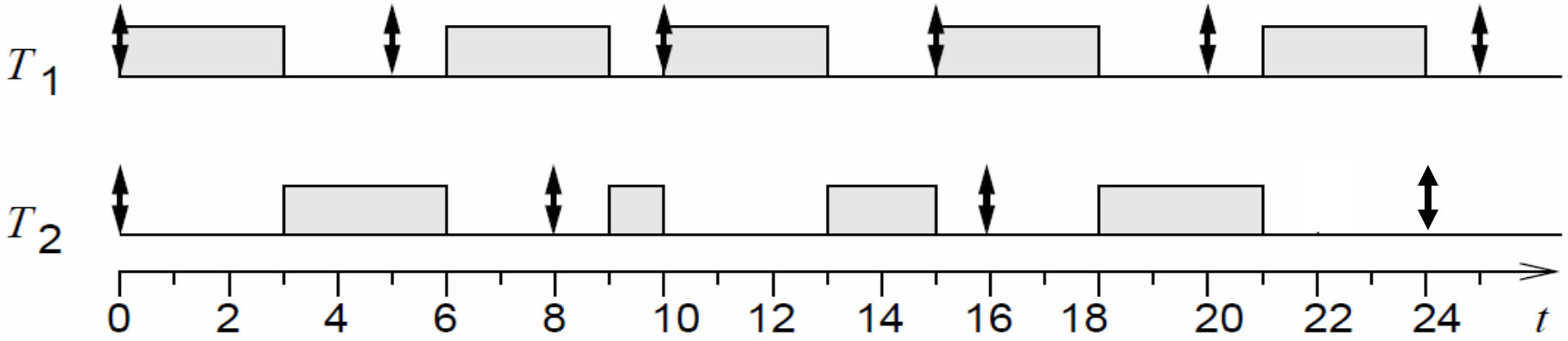
- ☞ Optimal for periodic scheduling
- ☞ EDF must be able to schedule the example in which RMS failed.

Comparison EDF/RMS

RMS:



EDF:



T_2 not preempted, due to its earlier deadline.

EDF: Properties

EDF requires dynamic priorities

- ☞ EDF cannot be used with an operating system just providing static priorities.

However, a recent paper (by Margull and Slomka) at DATE 2008 demonstrates how an OS with static priorities can be extended with a plug-in providing EDF scheduling
(key idea: delay tasks becoming ready if they shouldn't be executed under EDF scheduling.

Comparison RMS/EDF

	RMS	EDF
Priorities	Static	Dynamic
Works with OS with fixed priorities	**Yes**	**No***
Uses full computational power of processor	No, just up till $\mu = n(2^{1/n}-1)$	Yes
Possible to exploit full computational power of processor without provisioning for slack	**No**	**Yes**

* Unless the plug-in by Slomka et al. is added.

Sporadic tasks

If sporadic tasks were connected to interrupts, the execution time of other tasks would become very unpredictable.

- ☞ Introduction of a sporadic task server, periodically checking for ready sporadic tasks;
- ☞ Sporadic tasks are essentially turned into periodic tasks.

Dependent tasks

The problem of deciding whether or not a schedule exists for a set of dependent tasks and a given deadline is NP-complete in general [Garey/Johnson].

Strategies:

1. Add resources, so that scheduling becomes easier
2. Split problem into static and dynamic part so that only a minimum of decisions need to be taken at run-time.
3. Use scheduling algorithms from high-level synthesis

Summary

Periodic scheduling

- Rate monotonic scheduling
- EDF
- Dependent and sporadic tasks (briefly)

technische universität
dortmund

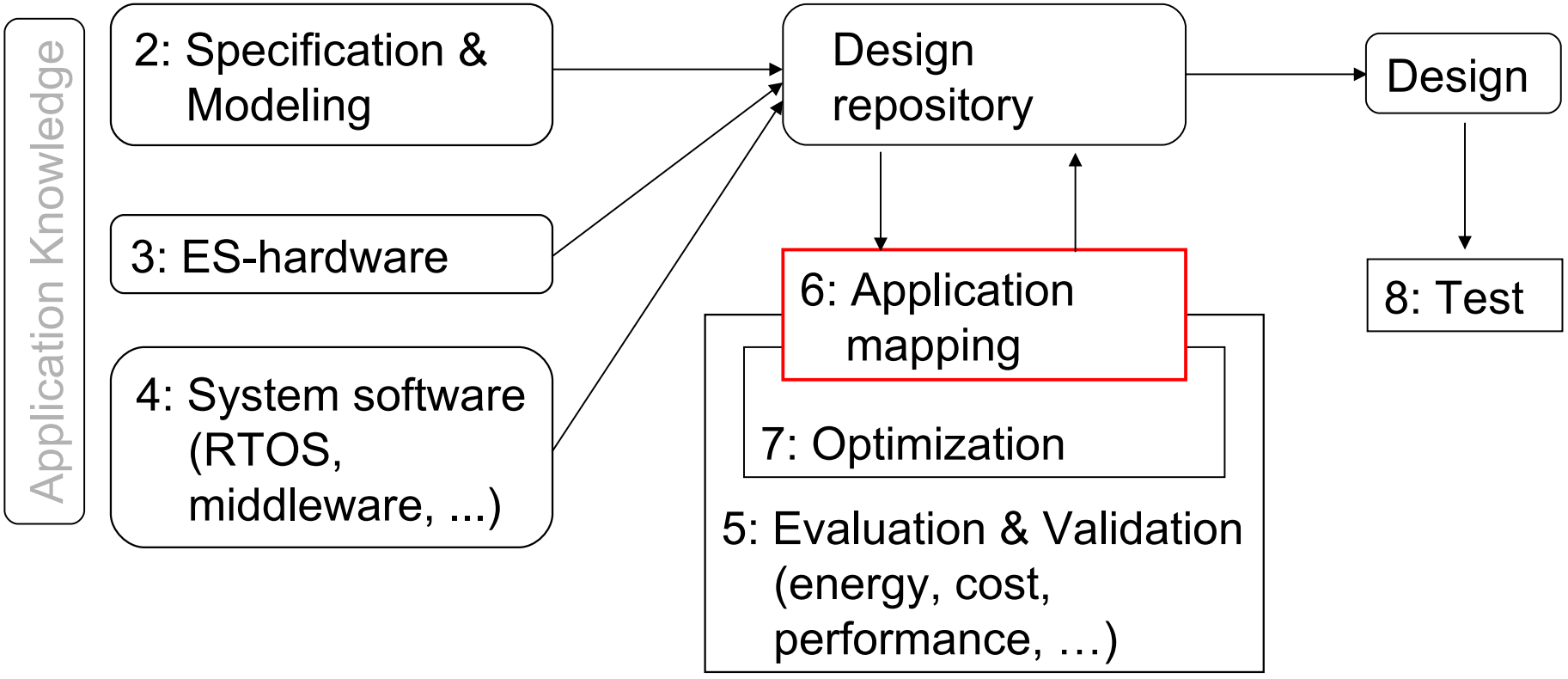
fakultät für informatik
informatik 12

Hardware / Software Partitioning

Peter Marwedel
TU Dortmund,
Informatik 12

2012 年 12 月 18 日

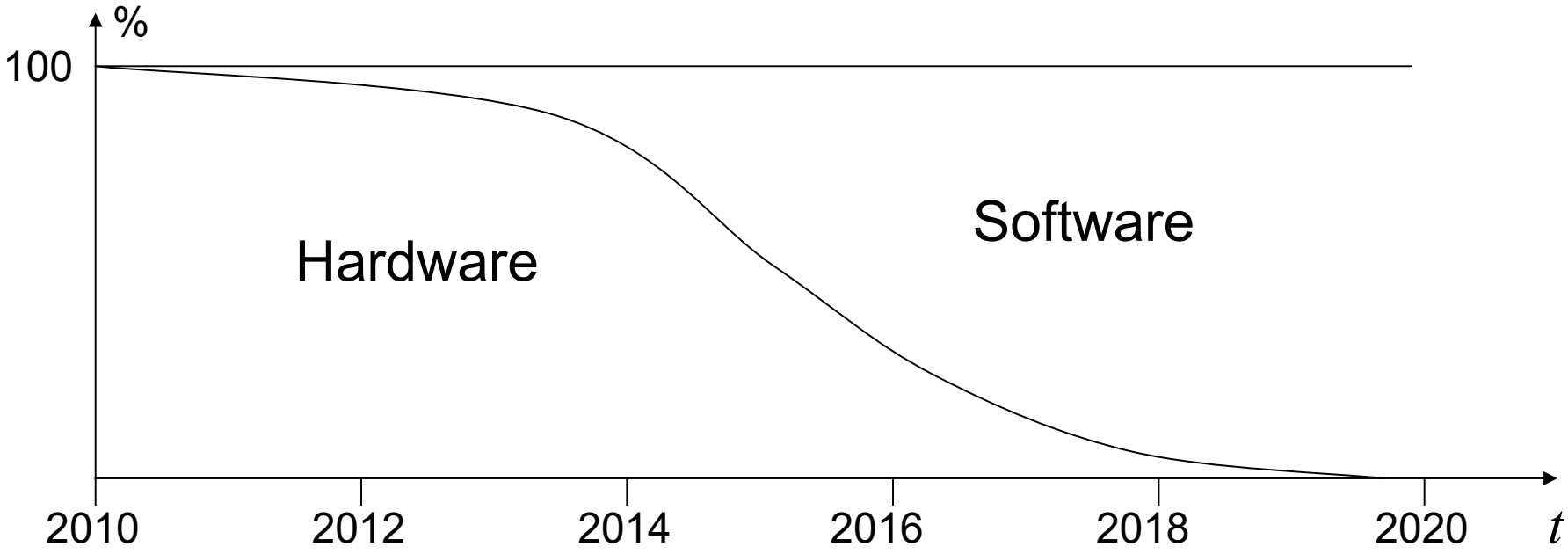
Structure of this course



Numbers denote sequence of chapters

Hardware/software partitioning

No need to consider special hardware in the future?

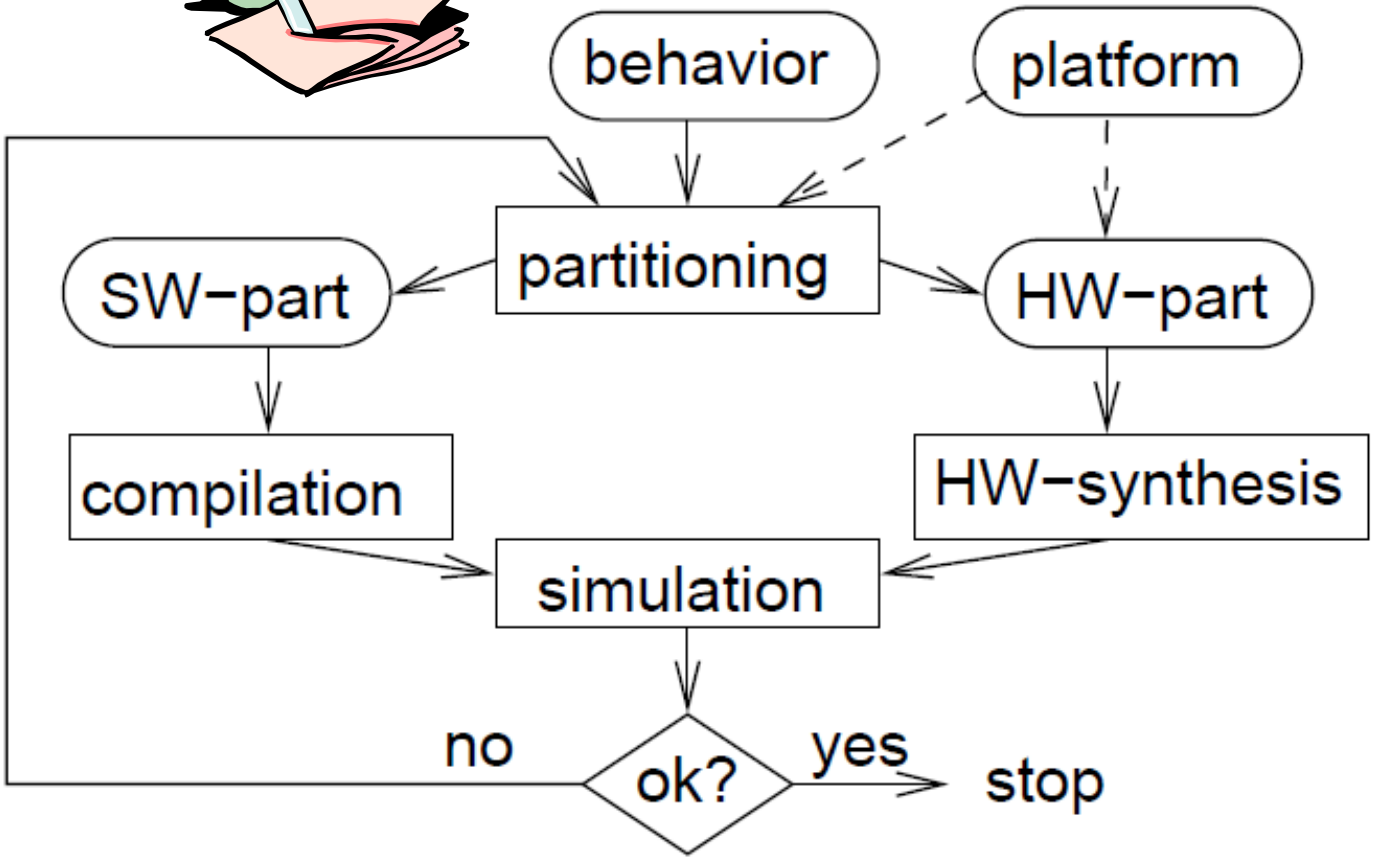


Correct for fixed functionality, but wrong in general: “By the time MPEG-n can be implemented in software, MPEG-n+1 has been invented” [de Man]

☞Functionality to be implemented in software or in hardware?

Functionality to be implemented in software or in hardware?

Decision based on hardware/ software partitioning, a special case of hardware/ software codesign.



Codesign Tool (COOL) as an example of HW/SW partitioning

Inputs to COOL:

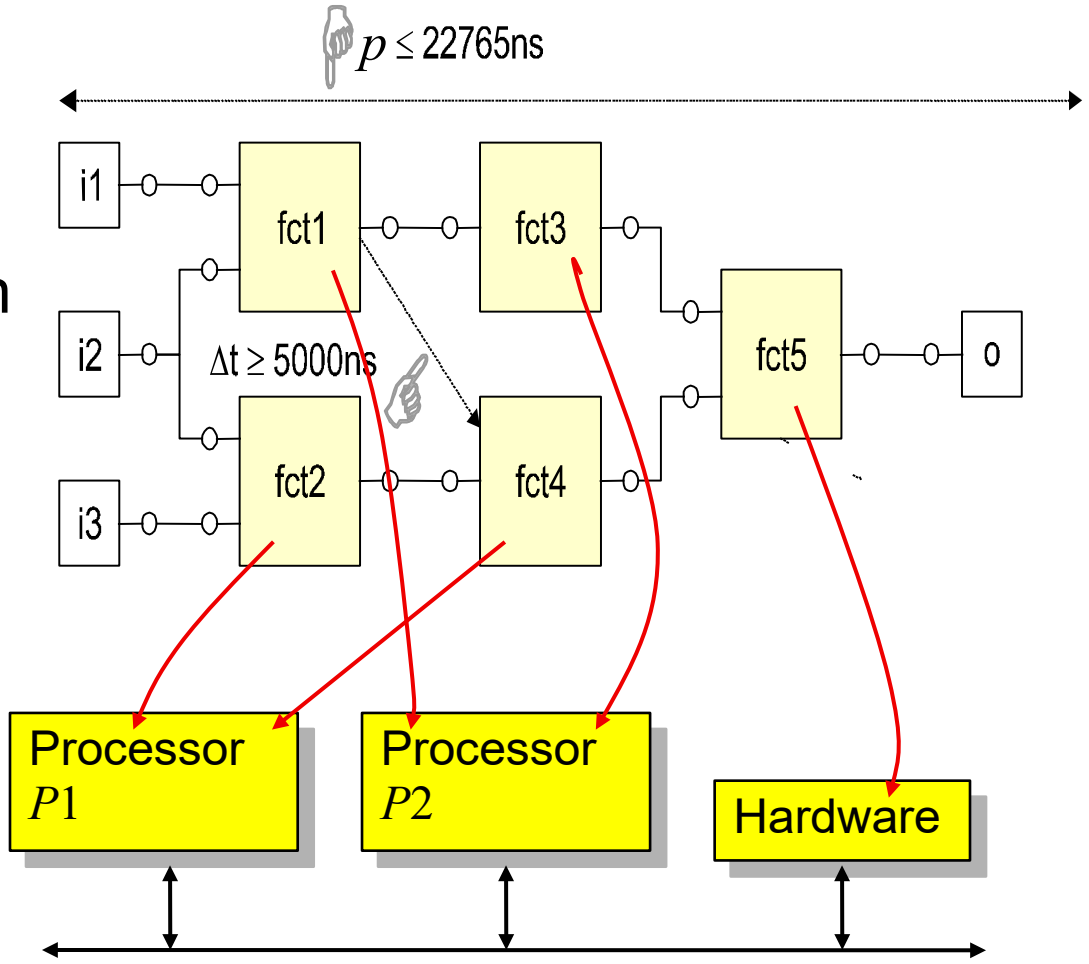
1. Target technology
2. Design constraints
3. Required behavior

ls12-www.cs.tu-dortmund.de/daes/de/forschung/hwsw-co-design/cool.html

Hardware/software codesign: approach

Specification

Mapping



Niemann, Hardware/Software Co-Design for Data Flow Dominated Embedded Systems, Kluwer Academic Publishers, 1998 (Comprehensive mathematical model)

Steps of the COOL partitioning algorithm (1)

1. **Translation of the behavior into an internal graph model**
2. **Translation of the behavior of each node from VHDL into C**
3. **Compilation**
   - All C programs compiled for the target processor,
   - Computation of the resulting program size,
   - estimation of the resulting execution time (simulation input data might be required)
4. **Synthesis of hardware components**
   ∀ leaf nodes, application-specific hardware is synthesized.
   High-level synthesis sufficiently fast.

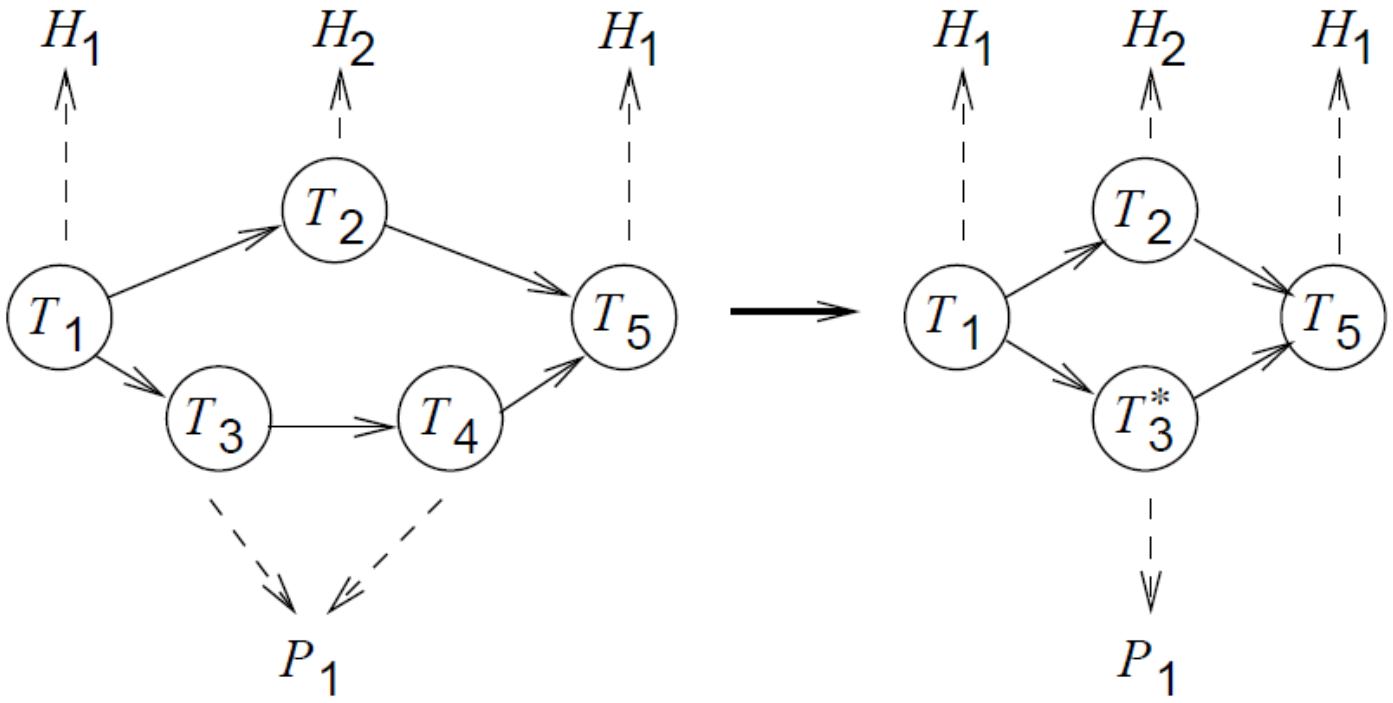
Steps of the COOL partitioning algorithm (2)

5. **Flattening of the hierarchy**
   - Granularity used by the designer is maintained.
   - Cost and performance information added to the nodes.
   - Precise information required for partitioning is pre-computed
6. **Generating and solving a mathematical model of the optimization problem**
   - Integer linear programming ILP model for optimization. Optimal with respect to the cost function (approximates communication time)

Steps of the COOL partitioning algorithm (3)

7. Iterative improvements

Adjacent nodes mapped to the same hardware component are now merged.



Steps of the COOL partitioning algorithm (4)

8. **Interface synthesis**

   After partitioning, the glue logic required for interfacing processors, application-specific hardware and memories is created.

An integer linear programming model for HW/SW partitioning

Notation:

- Index set V denotes task graph nodes.
- Index set L denotes task graph node **types**
  e.g. square root, DCT or FFT
- Index set M denotes hardware component **types**.
  e.g. hardware components for the DCT or the FFT.
- Index set J of hardware component instances
- Index set KP denotes processors.
  All processors are assumed to be of the same type

An ILP model for HW/SW partitioning

- $X_{v,m} = 1$ if node v is mapped to hardware component type $m \in M$ and 0 otherwise.
- $Y_{v,k} = 1$ if node v is mapped to processor $k \in KP$ and 0 otherwise.
- $NY_{l,k} = 1$ if at least one node of type l is mapped to processor $k \in KP$ and 0 otherwise.
- $Type$ is a mapping from task graph nodes to their types $Type : V \rightarrow L$
- The cost function accumulates the cost of hardware units

$$C = \text{cost}(processors) + \text{cost}(memories) + \text{cost}(application\ specific\ hardware)$$

Constraints

Operation assignment constraints

$$\forall v \in V : \sum_{m \in M} X_{v,m} + \sum_{k \in KP} Y_{v,k} = 1$$

All task graph nodes have to be mapped either in software or in hardware.

Variables are assumed to be integers.

Additional constraints to guarantee they are either 0 or 1:

$$\forall v \in V : \forall m \in M : X_{v,m} \leq 1$$

$$\forall v \in V : \forall k \in KP : Y_{v,k} \leq 1$$

Operation assignment constraints (2)

$$\forall l \in L, \forall v : Type(v) = c_l, \forall k \in KP : NY_{l,k} \geq Y_{v,k}$$

For all types l of operations and for all nodes v of this type: if v is mapped to some processor k, then that processor must implement the functionality of l.

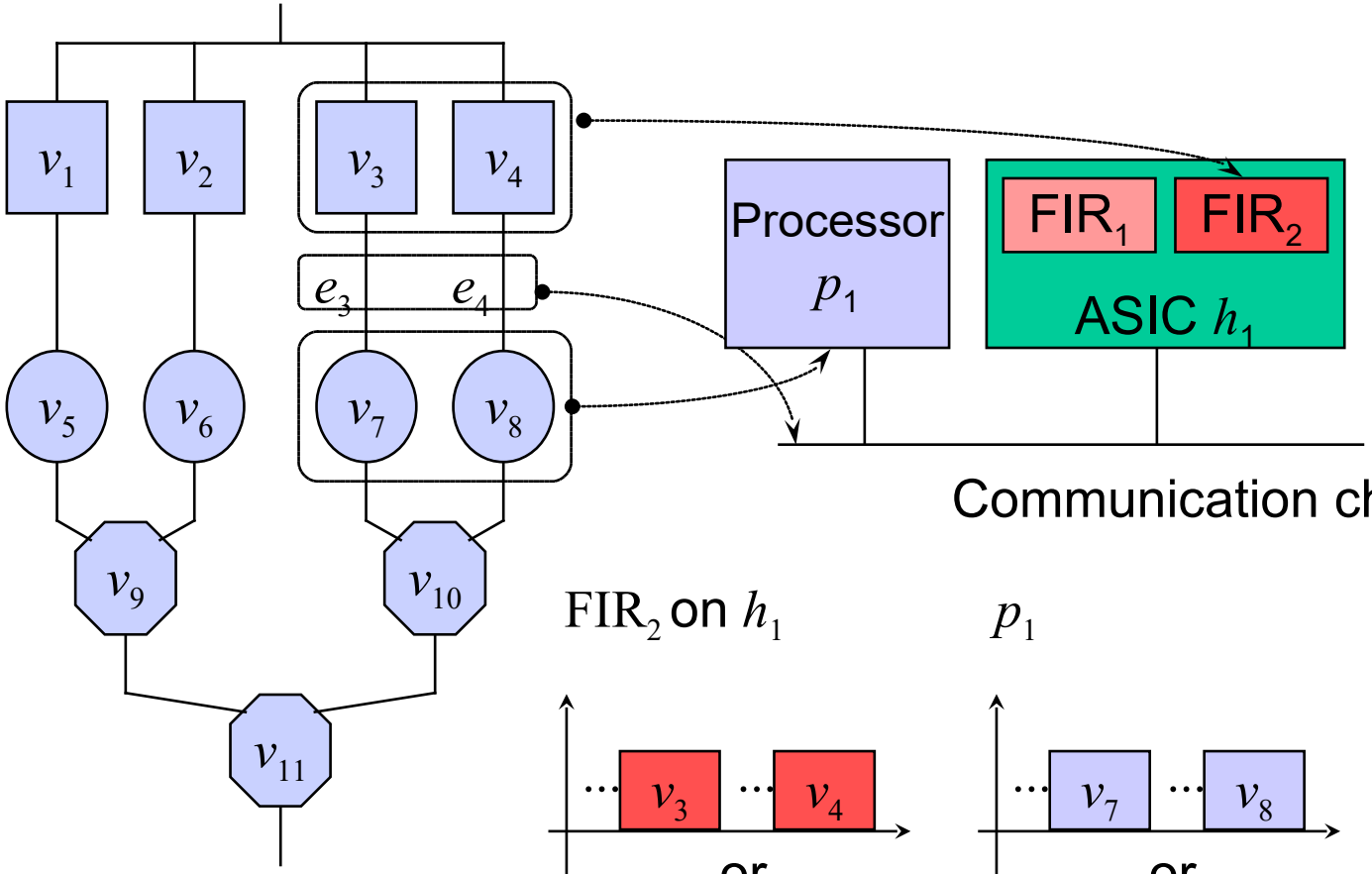
Decision variables must also be 0/1 variables:

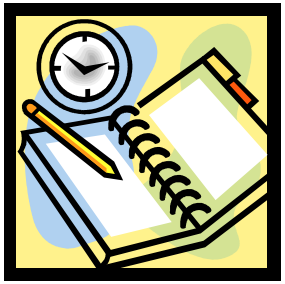
$$\forall l \in L, \forall k \in KP : NY_{l,k} \leq 1$$

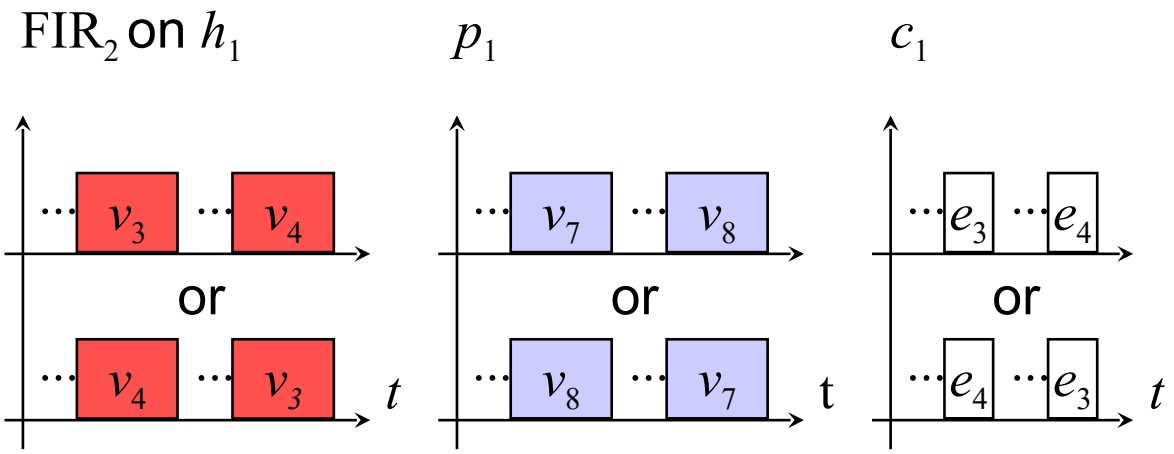
Resource & design constraints

- $\forall\ m \in M$, the cost (area) for components of type m is = sum of the costs of the components of that type. This cost should not exceed its maximum.
- $\forall\ k \in KP$, the cost for associated data storage area should not exceed its maximum.
- $\forall\ k \in KP$ the cost for storing instructions should not exceed its maximum.
- The total cost $(\Sigma_{m \in M})$ of HW components should not exceed its maximum
- The total cost of data memories $(\Sigma_{k \in KP})$ should not exceed its maximum
- The total cost instruction memories $(\Sigma_{k \in KP})$ should not exceed its maximum

Scheduling


v_1
v_2
v_3
v_4
e_3
e_4
v_5
v_6
v_7
v_8
v_9
v_{10}
v_{11}
Processor p_1
FIR$_1$
FIR$_2$
ASIC h_1
Communication channel c_1




FIR$_2$ on h_1
p_1
c_1
v_3 v_4
or
v_4 v_3
v_7 v_8
or
v_8 v_7
e_3 e_4
or
e_4 e_3
t
t
t

Scheduling / precedence constraints

- For all nodes v_{i1} and v_{i2} that are potentially mapped to the same processor or hardware component instance, introduce a binary decision variable $b_{i1,i2}$ with

  $b_{i1,i2} = 1$ if v_{i1} is executed before v_{i2} and

  $= 0$ otherwise.

  Define constraints of the type

  (end-time of v_{i1}) $\leq$ (start time of v_{i2}) if $b_{i1,i2}=1$ and

  (end-time of v_{i2}) $\leq$ (start time of v_{i1}) if $b_{i1,i2}=0$

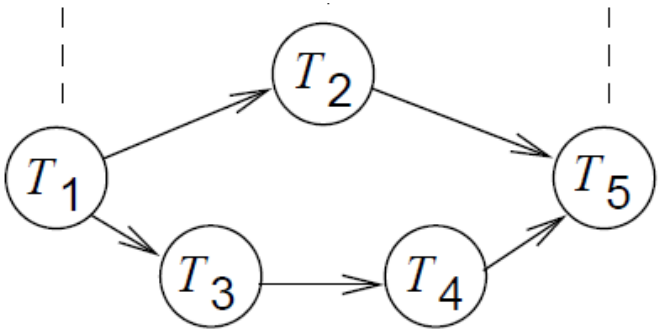
- Ensure that the schedule for executing operations is consistent with the precedence constraints in the task graph

- Approach fixes the order of execution

Other constraints

- **Timing constraints**
  These constraints can be used to guarantee that certain time constraints are met.
- Some less important constraints omitted ...

Example



HW types $H1$, $H2$ and $H3$ with costs of 20, 25, and 30.

Processors of type P.

Tasks $T1$ to $T5$.

Execution times:

T	$H1$	$H2$	$H3$	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100

Operation assignment constraints (1)

T	$H1$	$H2$	$H3$	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100

$$\forall v \in V : \sum_{m \in KM} X_{v,m} + \sum_{k \in KP} Y_{v,k} = 1$$

$X_{1,1} + Y_{1,1} = 1$ (task 1 mapped to $H1$ or to P)

$X_{2,2} + Y_{2,1} = 1$

$X_{3,3} + Y_{3,1} = 1$

$X_{4,3} + Y_{4,1} = 1$

$X_{5,1} + Y_{5,1} = 1$

Operation assignment constraints (2)

Assume types of tasks are l =1, 2, 3, 3, and 1.

$$\forall l \in L, \forall v : Type(v) = c_l, \forall k \in KP : NY_{l,k} \geq Y_{v,k}$$

$$NY_{1,1} \geq Y_{1,1}$$
$$NY_{2,1} \geq Y_{2,1}$$
$$NY_{3,1} \geq Y_{3,1}$$
$$NY_{3,1} \geq Y_{4,1}$$
$$NY_{1,1} \geq Y_{5,1}$$

Functionality 3 to be implemented on processor if node 4 is mapped to it.

Other equations

Time constraints leading to: Application specific hardware required for time constraints ≤ 100 time units.

T	$H1$	$H2$	$H3$	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100

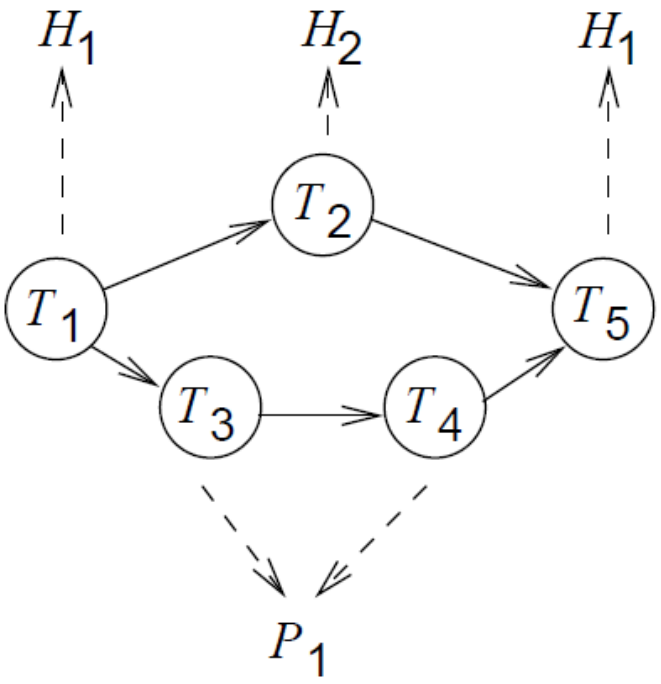
Cost function:

$$C = 20\ \#(H1) + 25\ \#(H2) + 30\ \#\ (H3) + \text{cost}(processor) + \text{cost}(memory)$$

Result

For a time constraint of 100 time units and $\text{cost}(P) < \text{cost}(H3)$:

T	$H1$	$H2$	$H3$	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100



Solution (educated guessing) :

$T1 \rightarrow H1$

$T2 \rightarrow H2$

$T3 \rightarrow P$

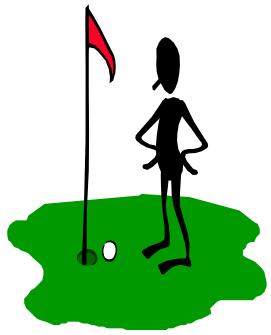
$T4 \rightarrow P$

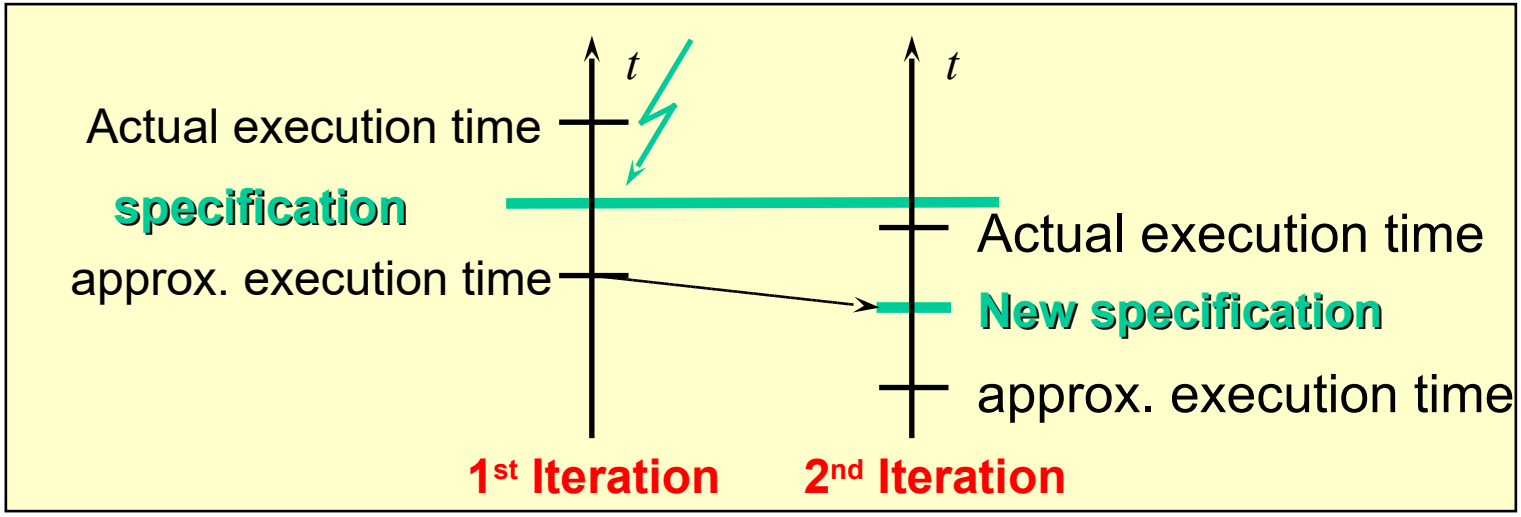
$T5 \rightarrow H1$

Separation of scheduling and partitioning

Combined scheduling/partitioning very complex

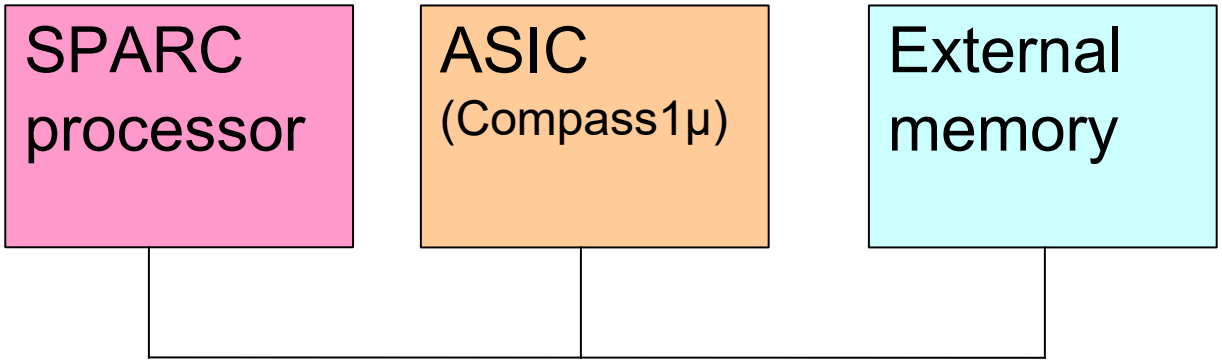
- ☞ Heuristic: Compute estimated schedule
  - Perform partitioning for estimated schedule
  - Perform final scheduling
  - If final schedule does not meet time constraint, go to 1 using a reduced overall timing constraint.





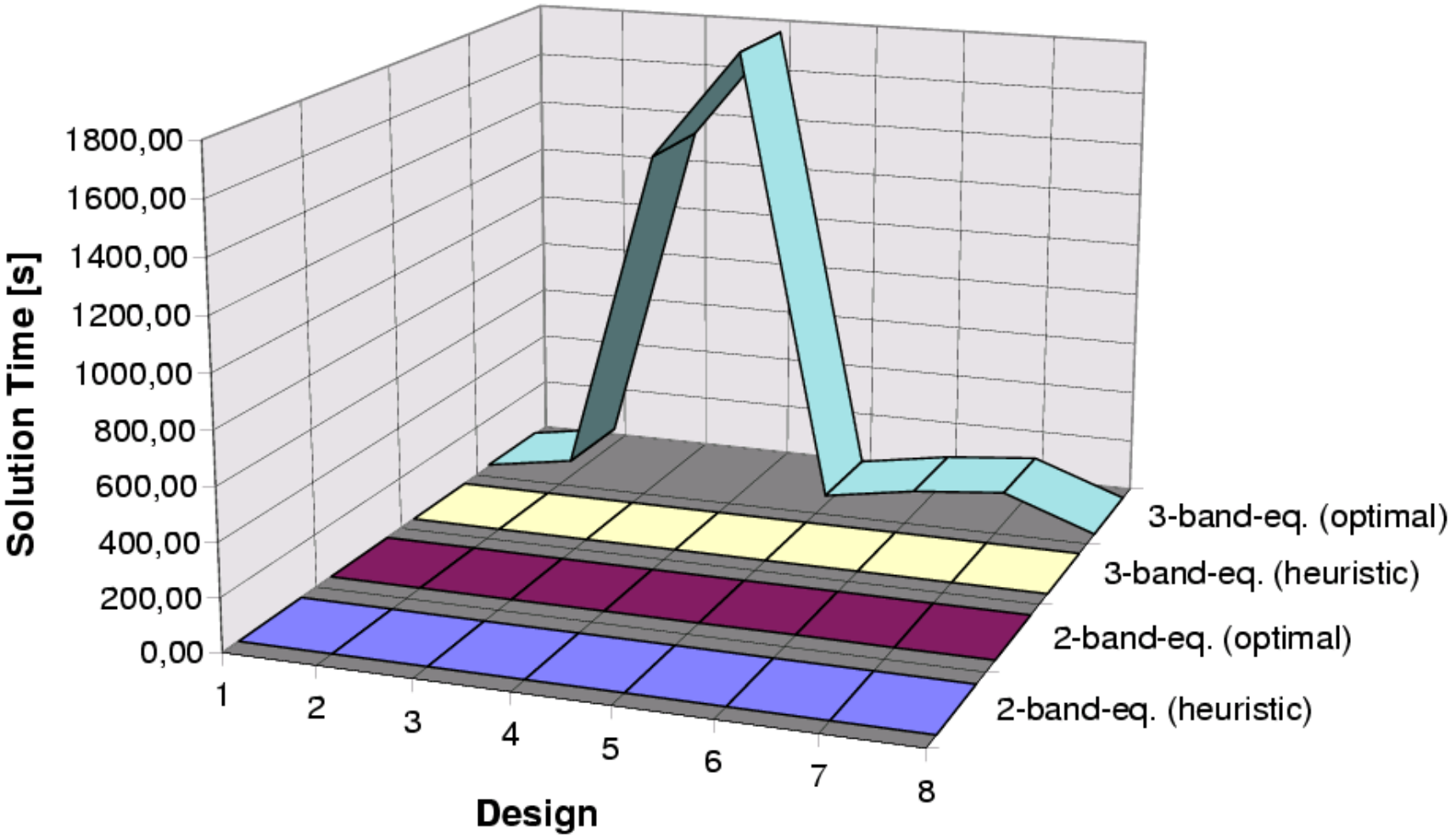
Application example

Audio lab (mixer, fader, echo, equalizer, balance units); slow SPARC processor
1µ ASIC library
Allowable delay of 22.675 µs (~ 44.1 kHz)



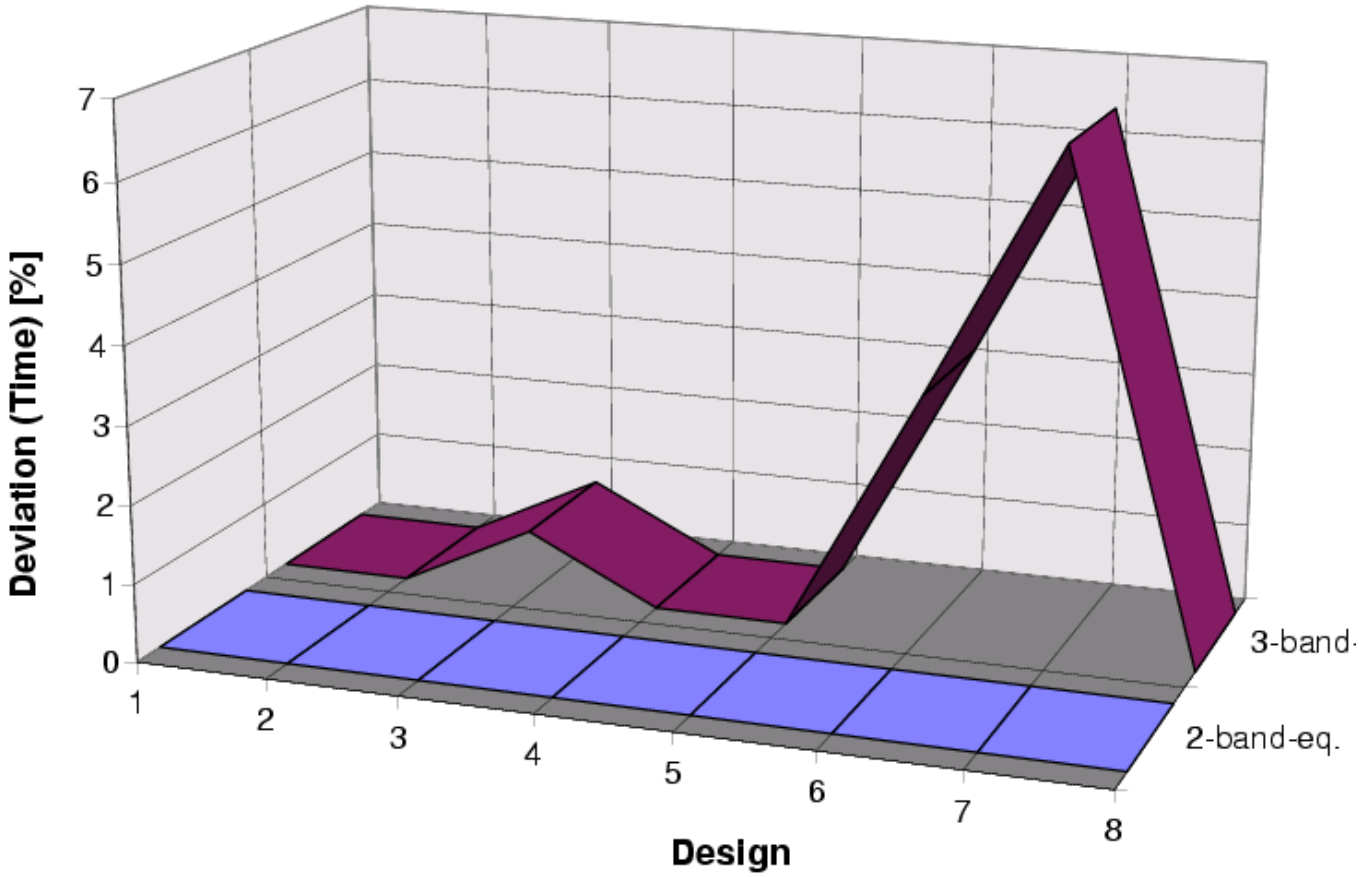
Outdated technology; just a proof of concept.

Running time for COOL optimization



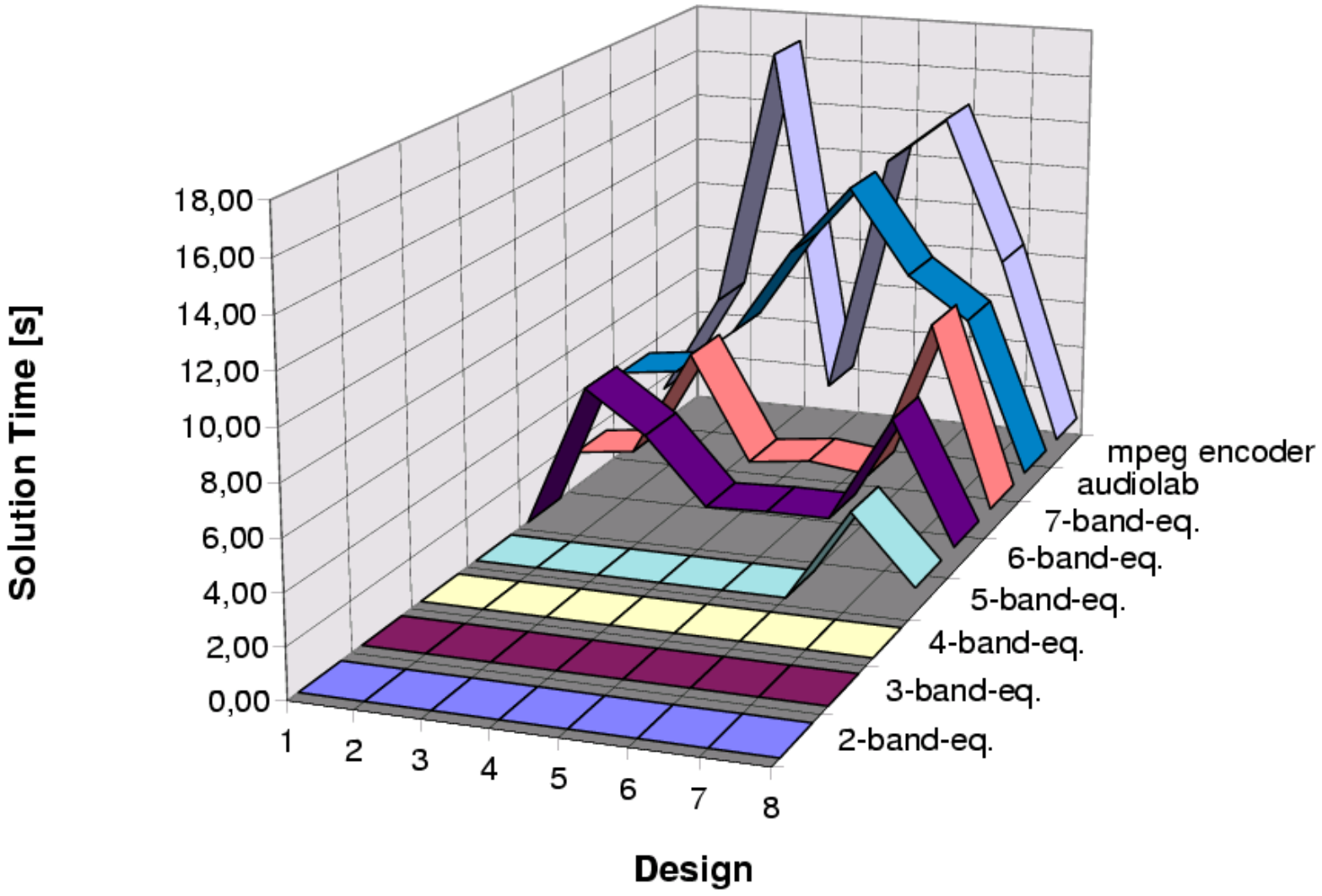
☞ Only simple models can be solved optimally.

Deviation from optimal design

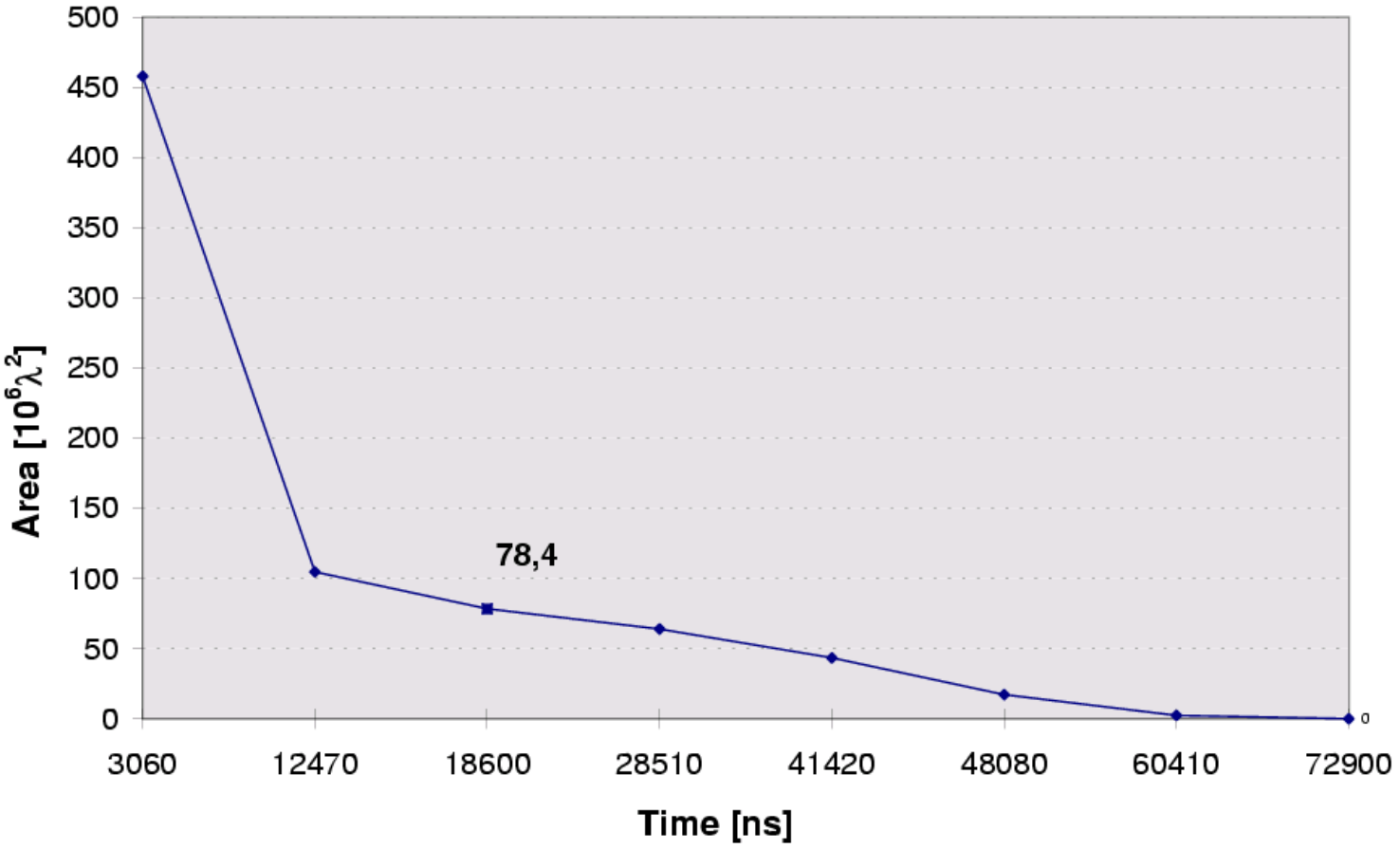


☞ Hardly any loss in design quality.

Running time for heuristic



Design space for audio lab



Everything in software	72.9 µs,	0 λ^2
Everything in hardware	3.06 µs,	457.9x10^6 λ^2
Lowest cost for given sample rate	18.6 µs,	78.4x10^6 λ^2

Positioning of COOL

COOL approach:

- shows that a formal model of hardware/SW codesign is beneficial; IP modeling can lead to useful implementation even if optimal result is available only for small designs.

Other approaches for HW/SW partitioning:

- starting with everything mapped to hardware; gradually moving to software as long as timing constraint is met.
- starting with everything mapped to software; gradually moving to hardware until timing constraint is met.
- Binary search.

HW/SW partitioning in the context of mapping applications to processors

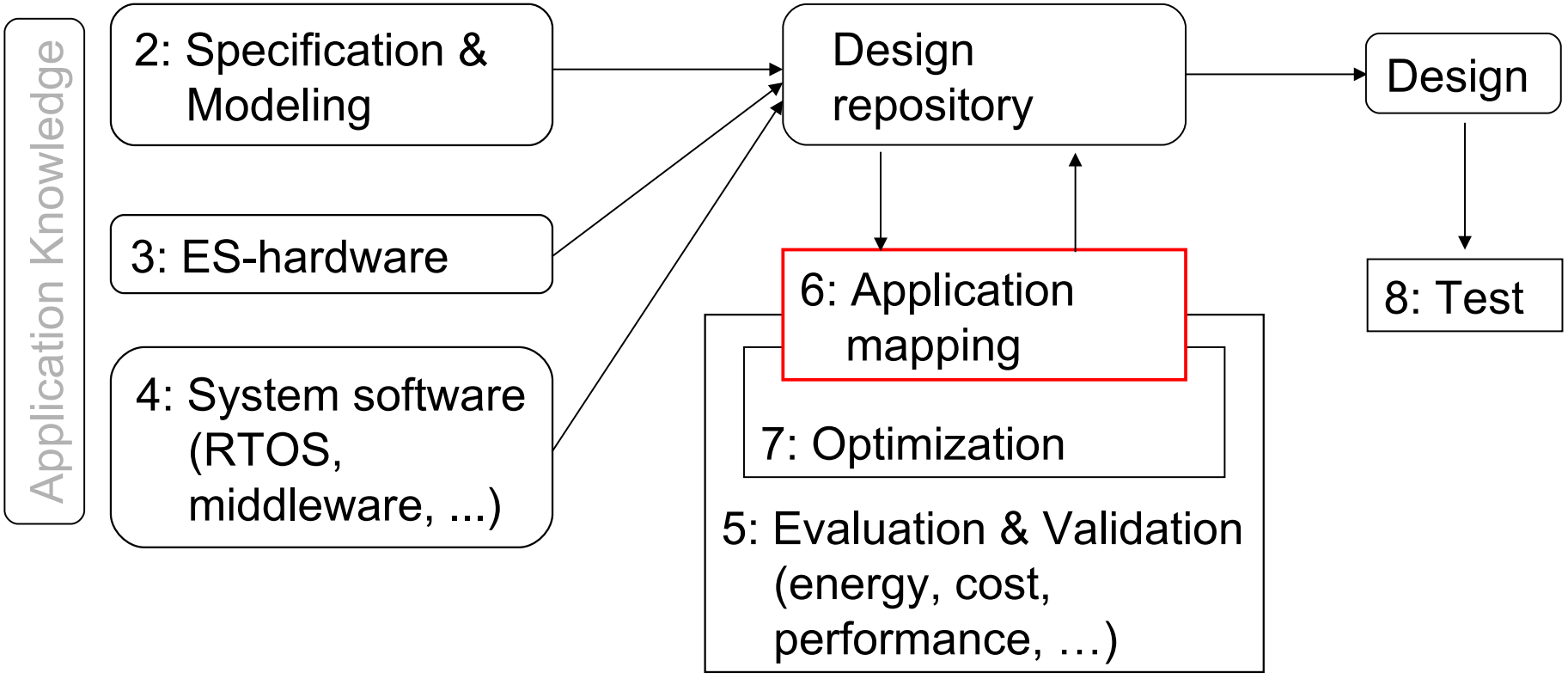
- Handling of heterogeneous systems
- Handling of task dependencies
- Considers of communication (at least in COOL)
- Considers memory sizes etc (at least in COOL)
- For COOL: just homogeneous processors
- No link to scheduling theory

Mapping of Applications to Multi-Processor Systems

Peter Marwedel
TU Dortmund,
Informatik 12

2014 年 01 月 17 日

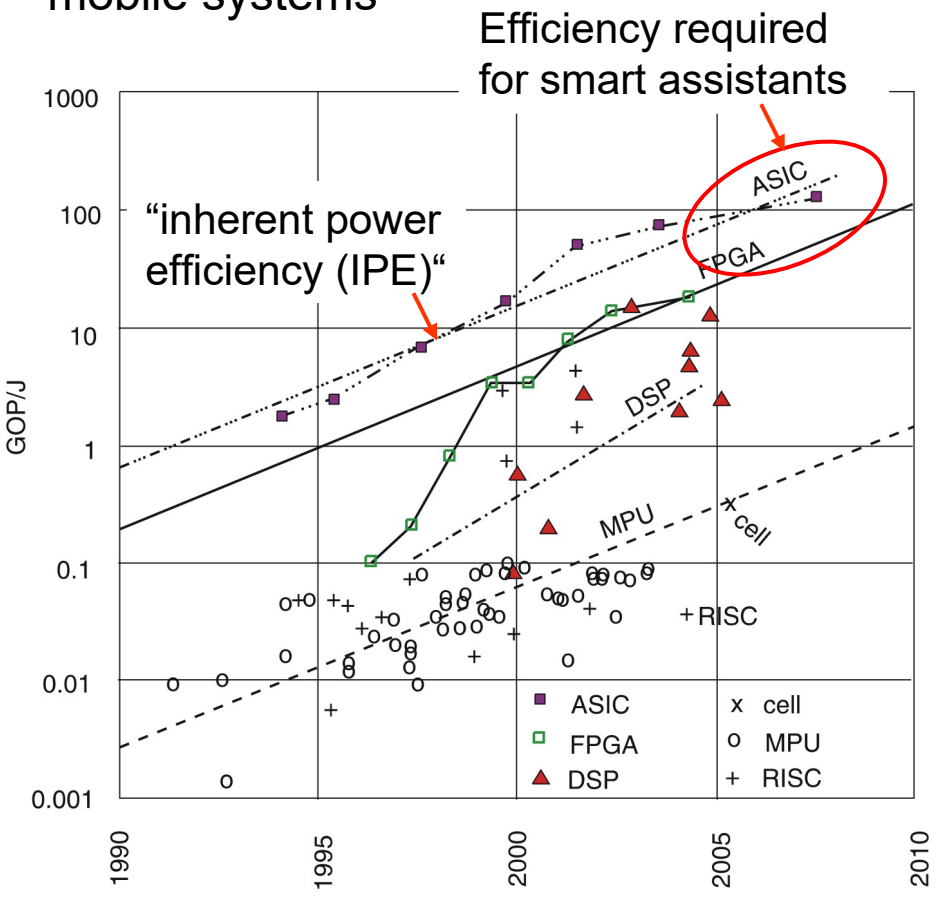
Structure of this course



Numbers denote sequence of chapters

The need to support heterogeneous architectures

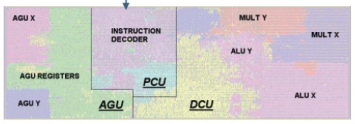
Energy efficiency a key constraint, e.g. for mobile systems



© Hugo De Man/Philips, 2007

Unconventional architectures close to IPE

Retargetable C compiler

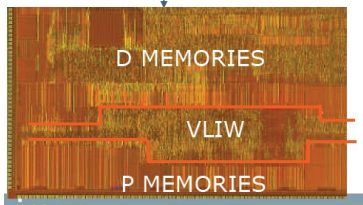


Courtesy: Philips-Target Compilers

Coolflux Audio ASIP

130 nm 0.9V 0.32mm² 24bit
2.0 mW MP3 incl. SRAMs
42 MOPS/mW (~1/4 IPE)

Retargetable C compiler

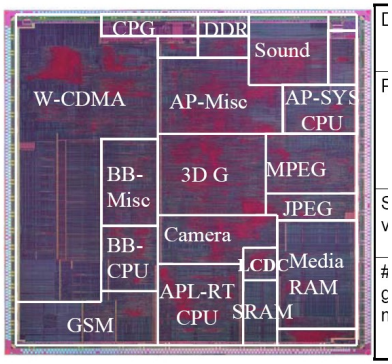


Courtesy: SiliconHive

41 Issue VLIW for SDR

130 nm 1.2V 6.5mm² 16 bit
30 operations / cycle (OFDM)
150 MHz 190mW (incl SRAMs)
24 GOPS/W (~ 1/5 IPE)

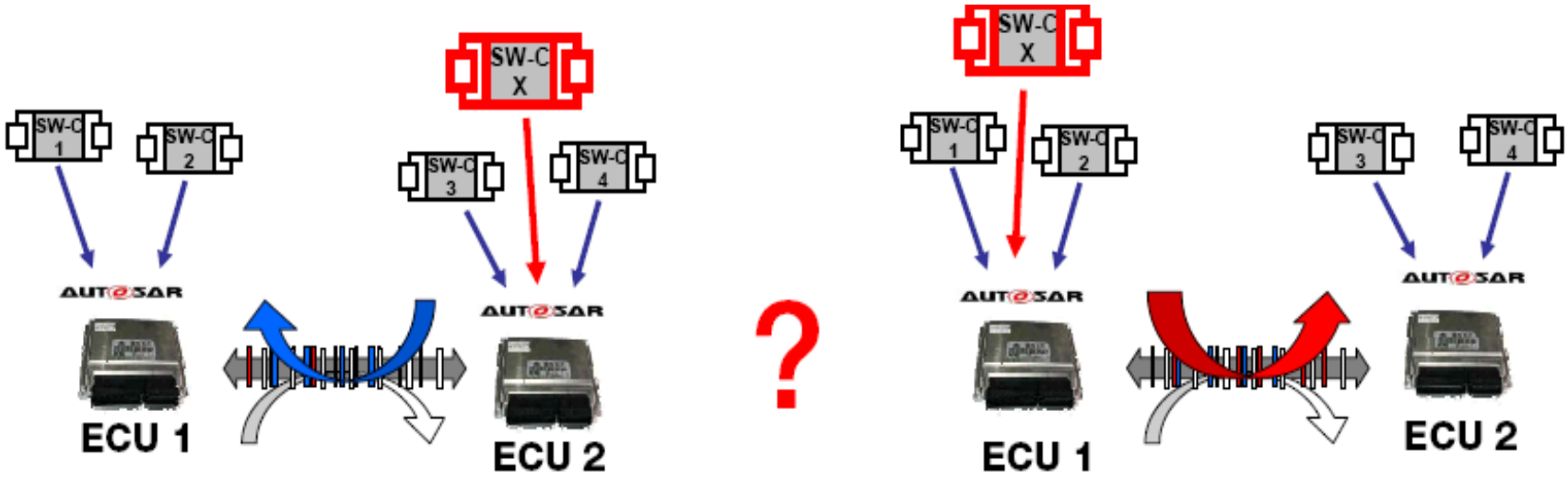
SH-MobileG1: Chip Overview



© Renesas, MPSoC'07

☞ How to map to these architectures?

Practical problem in automotive design



- Evaluate alternatives („what if ?“)
  - Mapping
  - Scheduling
  - Communication

- **Early**
- **Quickly**
- **Cost-efficient**

Which processor should run the software?

A Simple Classification

Architecture fixed Auto-parallelizing	Fixed Architecture	Architecture to be designed
Starting from given task graph	Map to CELL, Hopes, Qiang XU (HK) Simunic (UCSD)	COOL codesign tool; EXPO/SPEA2 SystemCodesigner
Auto-parallelizing	Mnemee (Dortmund) Franke (Edinburgh) MAPS	Daedalus

Example: System Synthesis

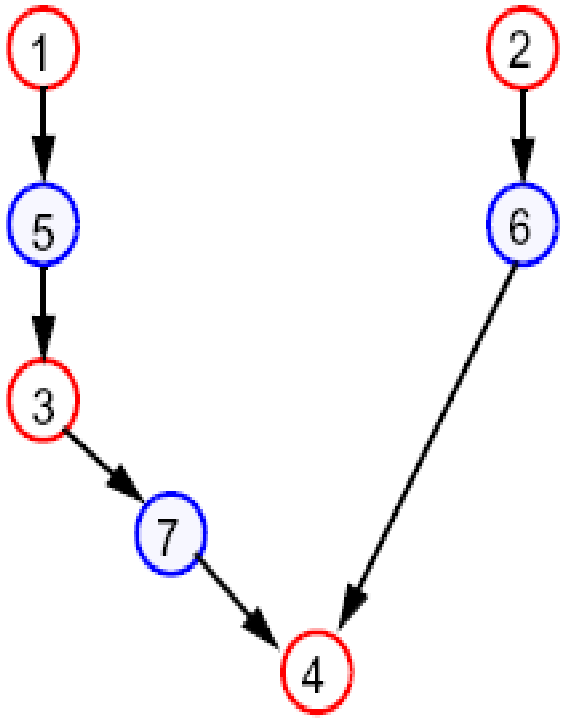
Given:

Goal:

Objectives: cost, latency, power consumption

Basic Model – Problem Graph

Problem graph $G_P(V_P,E_P)$:

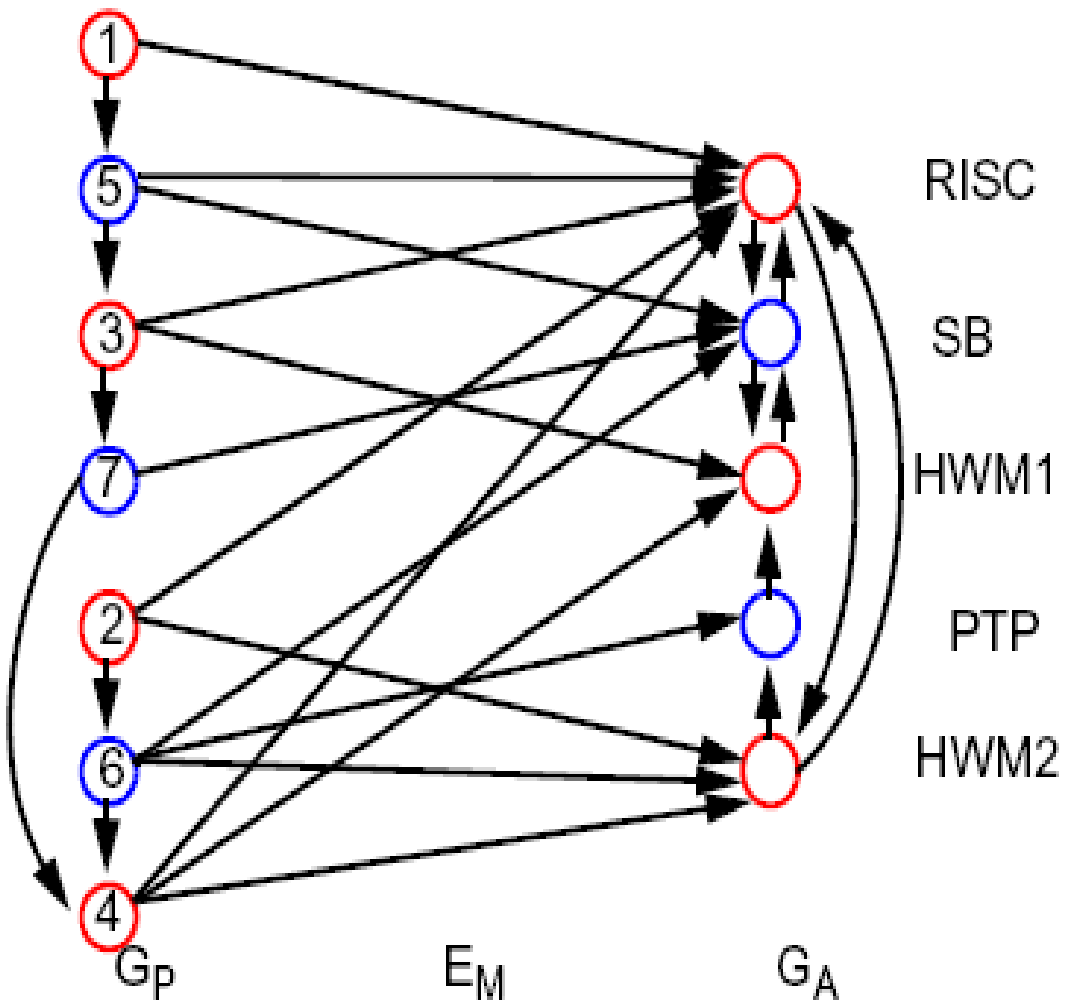


Interpretation:

- V_P consists of functional nodes V_P^f (task, procedure) and communication nodes V_P^c .
- E_P represent data dependencies

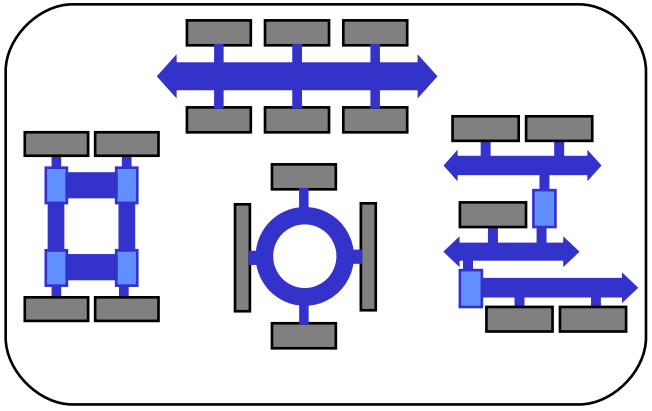
Basic Model: Specification Graph

Definition: A specification graph is a graph $G_S=(V_S,E_S)$ consisting of a problem graph G_P, an architecture graph G_A, and edges E_M. In particular, $V_S=V_P \cup V_A$, $E_S=E_P \cup E_A \cup E_M$

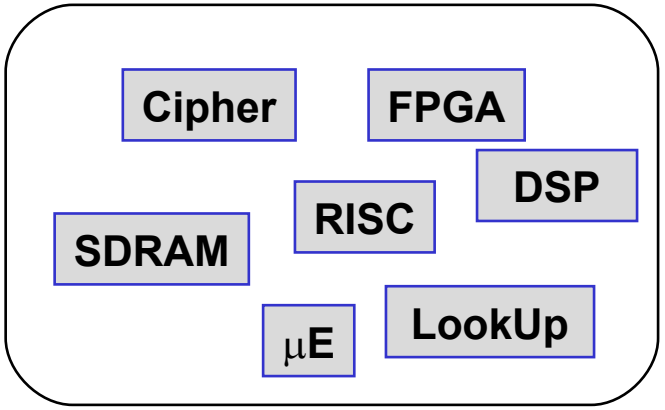


Design Space

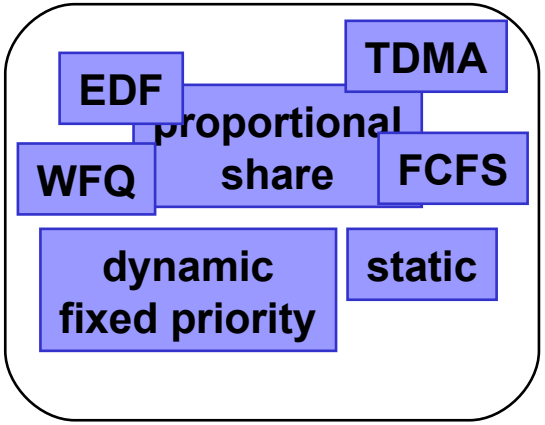
Communication Templates



Computation Templates

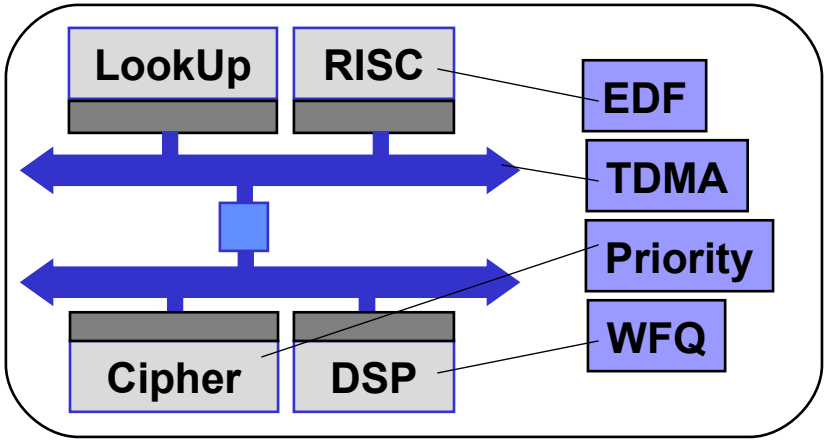


Scheduling/Arbitration

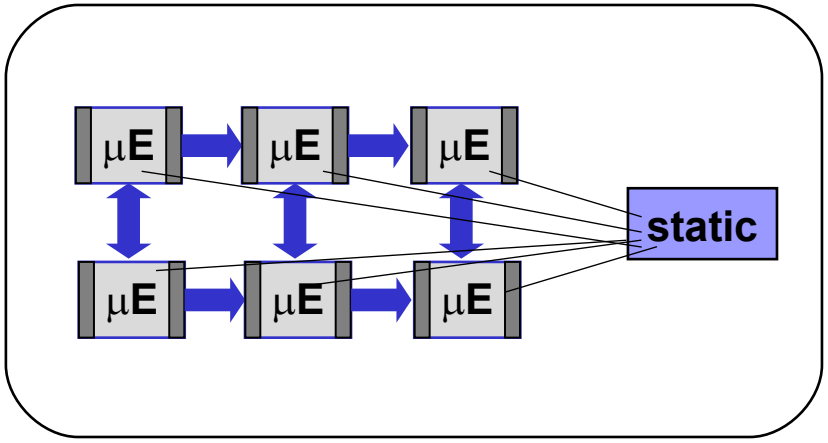


Which architecture is better suited for our application?

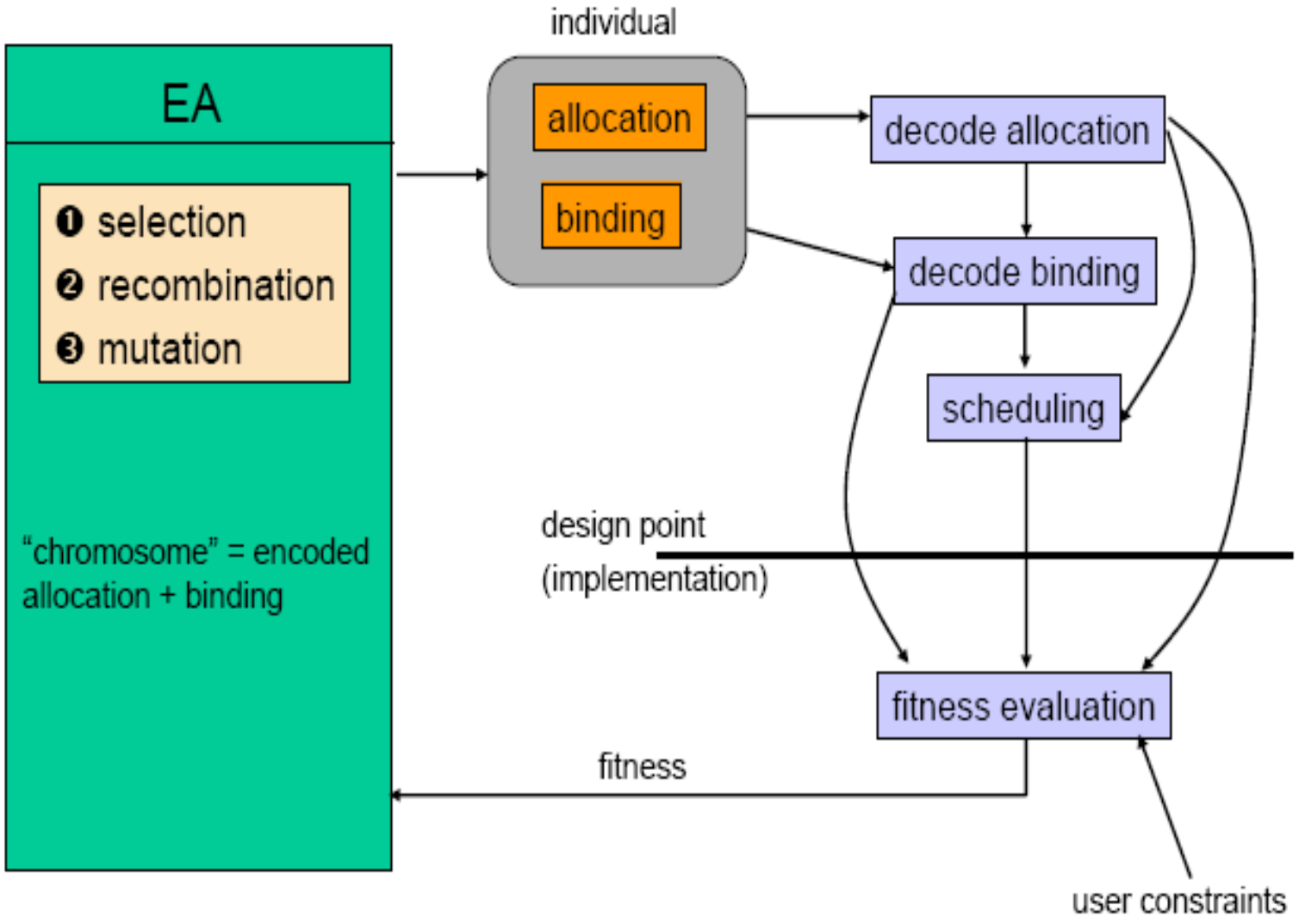
Architecture # 1



Architecture # 2



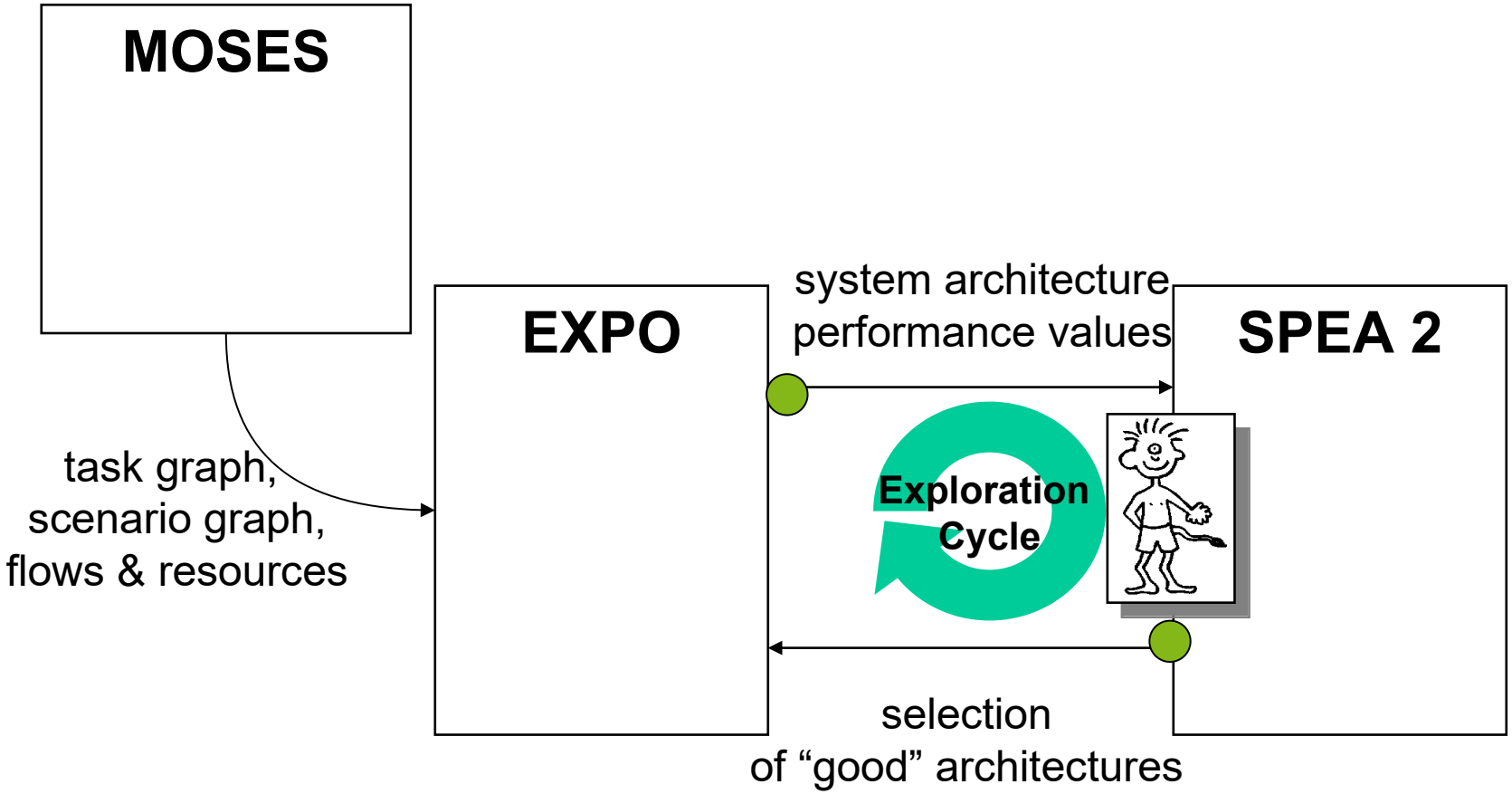
Evolutionary Algorithms for Design Space Exploration (DSE)



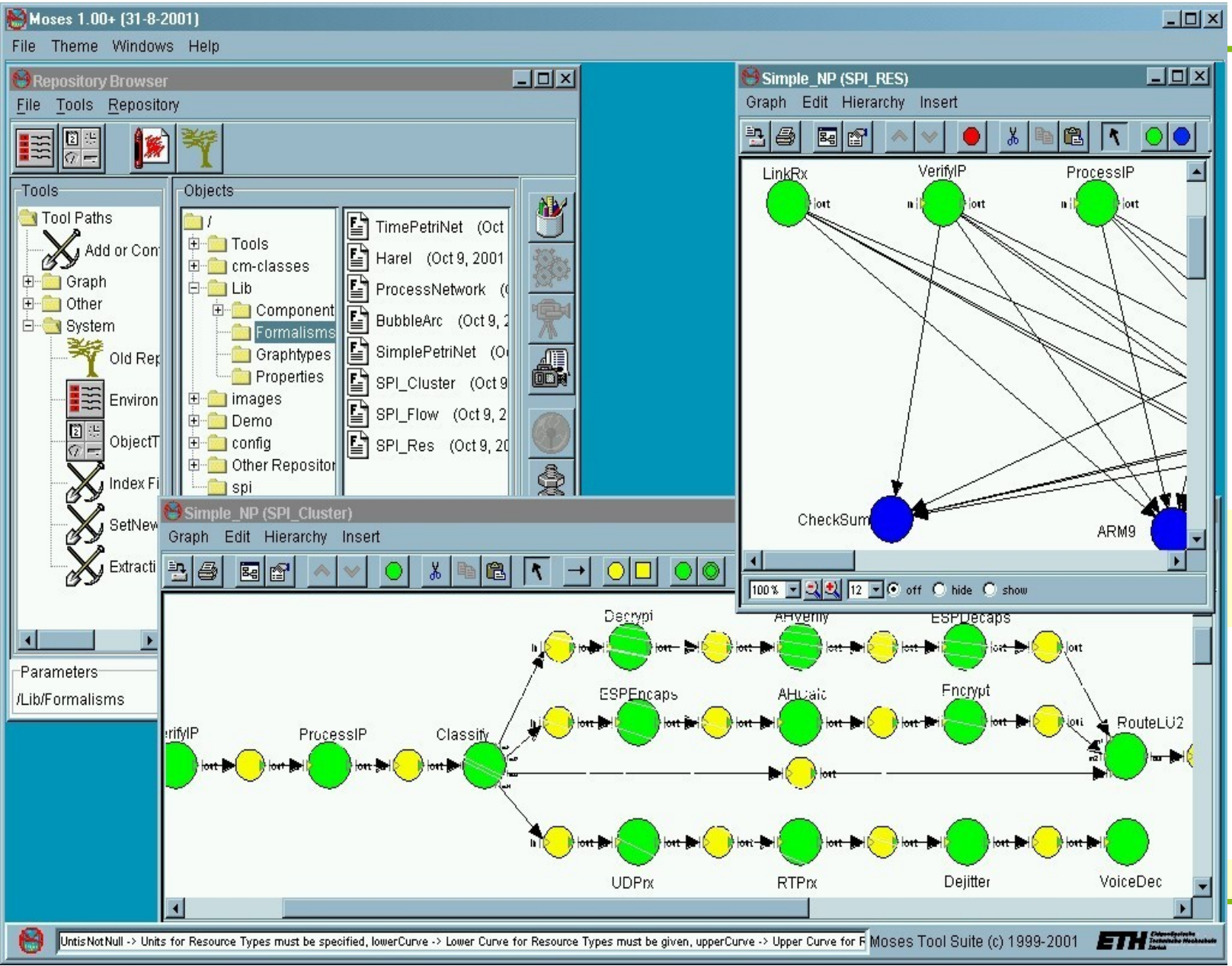
Challenges

- Encoding of (allocation+binding)
  - simple encoding
    - eg. one bit per resource, one variable per binding
    - easy to implement
    - many infeasible partitionings
  - encoding + repair
    - eg. simple encoding and modify such that for each $v_p \in V_P$ there exists at least one $v_a \in V_A$ with a $\beta(v_p) = v_a$
    - reduces number of infeasible partitionings
- Generation of the initial population, mutation
- Recombination

EXPO – Tool architecture (1)

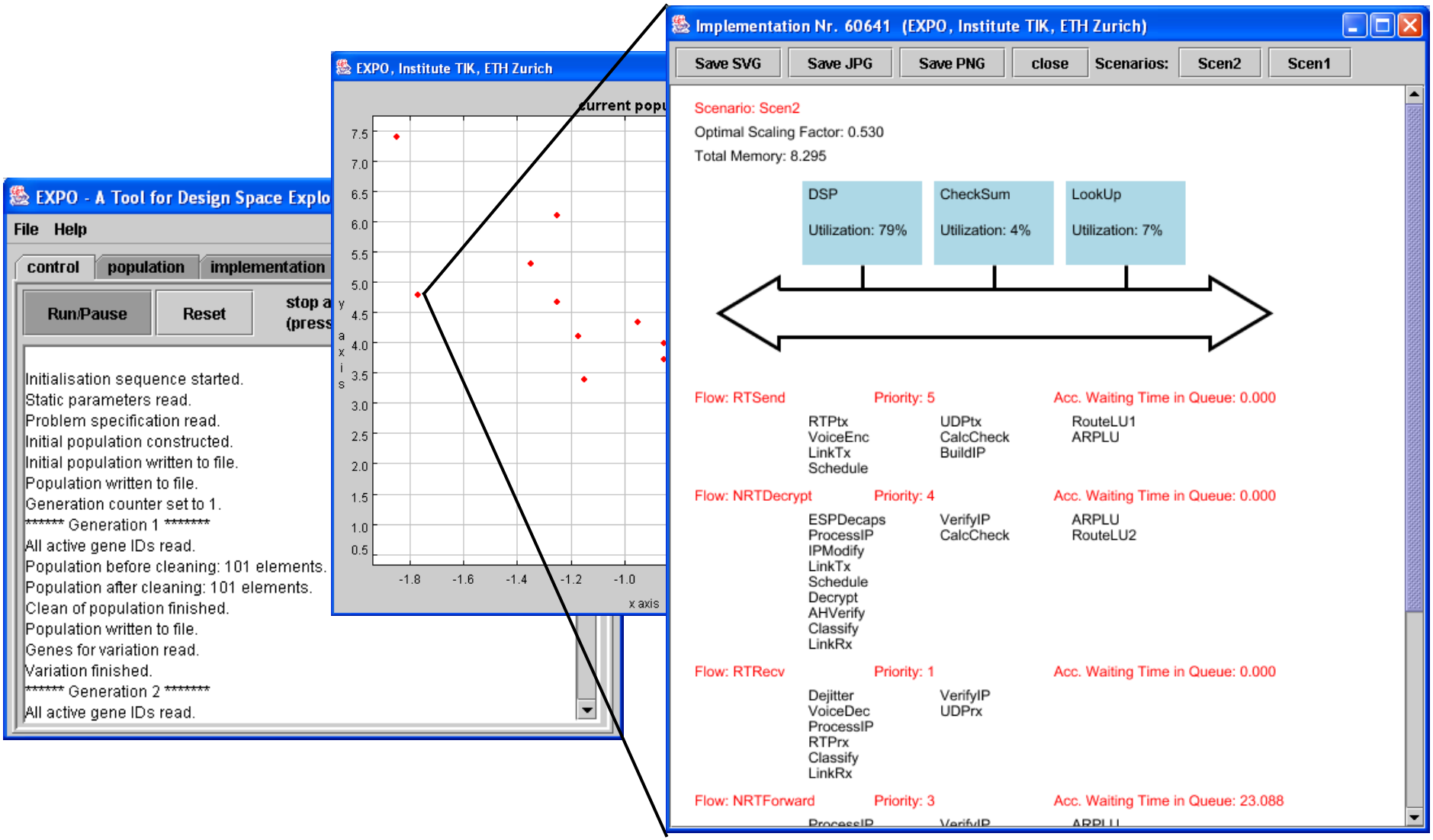


EXPO – Tool architecture (2)



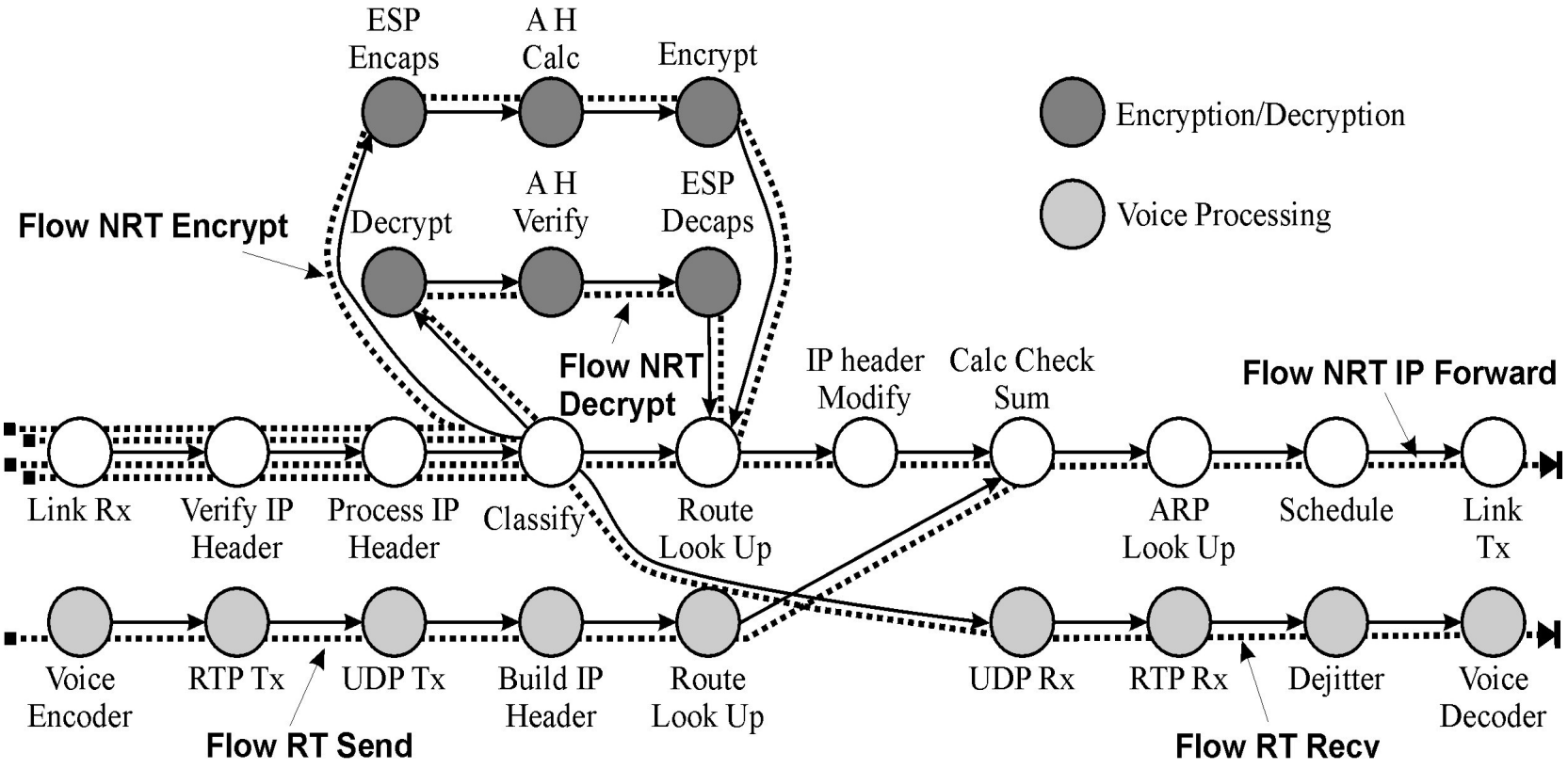
Tool available online
https://sop.tik.ee.ethz.ch/pisa/variators/expo/documentation.html

EXPO – Tool (3)


EXPO - A Tool for Design Space Explo
File Help
control population implementation
Run/Pause Reset
Initialisation sequence started.
Static parameters read.
Problem specification read.
Initial population constructed.
Initial population written to file.
Population written to file.
Generation counter set to 1.
****** Generation 1 *******
All active gene IDs read.
Population before cleaning: 101 elements.
Population after cleaning: 101 elements.
Clean of population finished.
Population written to file.
Genes for variation read.
Variation finished.
****** Generation 2 *******
All active gene IDs read.
EXPO, Institute TIK, ETH Zurich
y axis
x axis
Implementation Nr. 60641 (EXPO, Institute TIK, ETH Zurich)
Save SVG Save JPG Save PNG close Scenarios: Scen2 Scen1
Scenario: Scen2
Optimal Scaling Factor: 0.530
Total Memory: 8.295
DSP
Utilization: 79%
CheckSum
Utilization: 4%
LookUp
Utilization: 7%
Flow: RTSend Priority: 5 Acc. Waiting Time in Queue: 0.000
RTPtx VoiceEnc LinkTx Schedule
UDPtx CalcCheck BuildIP
RouteLU1 ARPLU
Flow: NRTDecrypt Priority: 4 Acc. Waiting Time in Queue: 0.000
ESPDecaps ProcessIP IPModify LinkTx Schedule Decrypt AHVerify Classify LinkRx
VerifyIP CalcCheck
ARPLU RouteLU2
Flow: RTRecv Priority: 1 Acc. Waiting Time in Queue: 0.000
Dejitter VoiceDec ProcessIP RTPrx Classify LinkRx
VerifyIP UDPrx
Flow: NRTForward Priority: 3 Acc. Waiting Time in Queue: 23.088

Application Model

Example of a simple stream processing task structure:



Summary

- Clear trend toward multi-processor systems
  for embedded systems, there exists a large design space
- Using architecture **crucially** depends on **mapping tools**
- Mapping applications onto heterogeneous MP systems needs allocation (if hardware is not fixed), binding of tasks to resources, scheduling
- Two criteria for classification
  - Fixed / flexible architecture
  - Auto parallelizing / non-parallelizing
- Introduction to proposed Mnemee tool chain

Evolutionary / Genetic algorithms currently the best choice

technische universität
dortmund

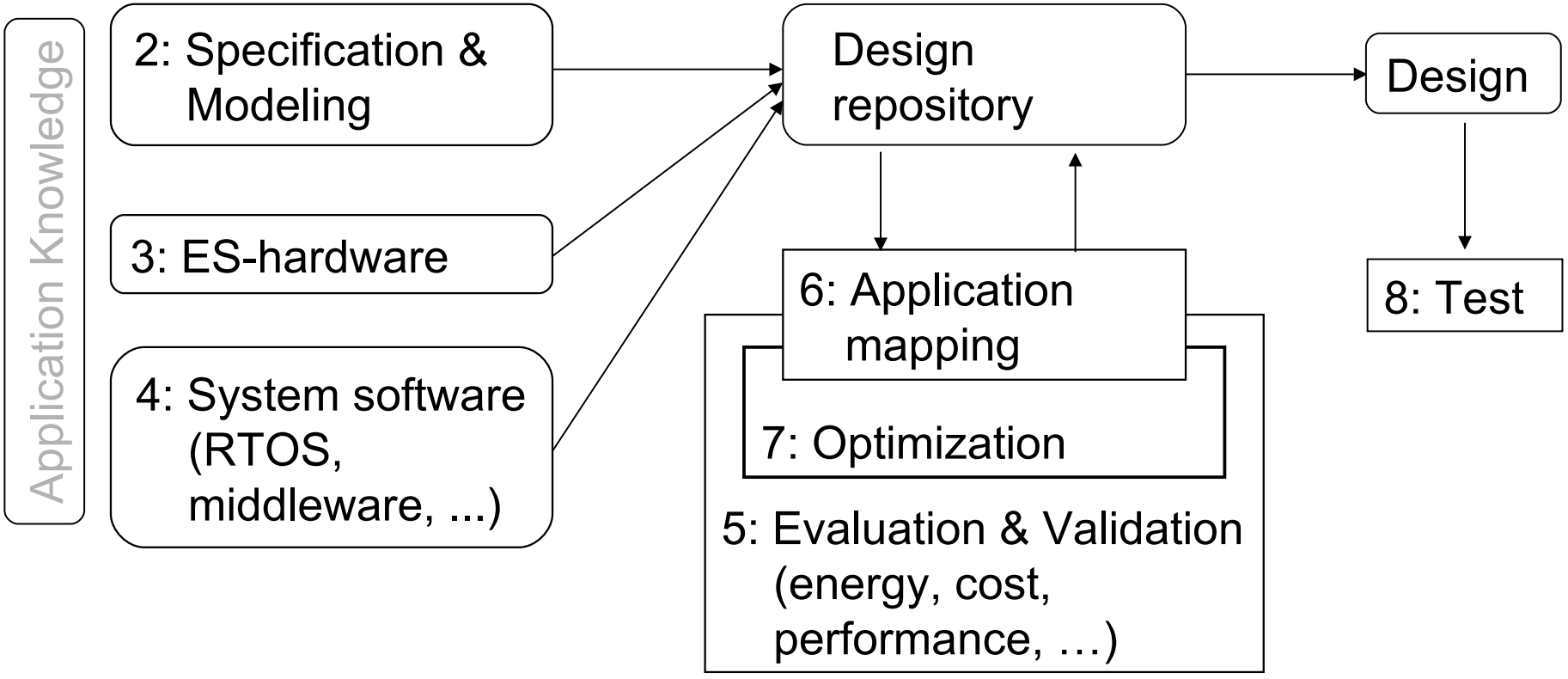
fakultät für informatik
informatik 12

Optimizations
- Compilation for Embedded Processors -

Peter Marwedel
TU Dortmund,
Informatik 12

2014 年 01 月 17 日

Structure of this course



Numbers denote sequence of chapters

Task-level concurrency management

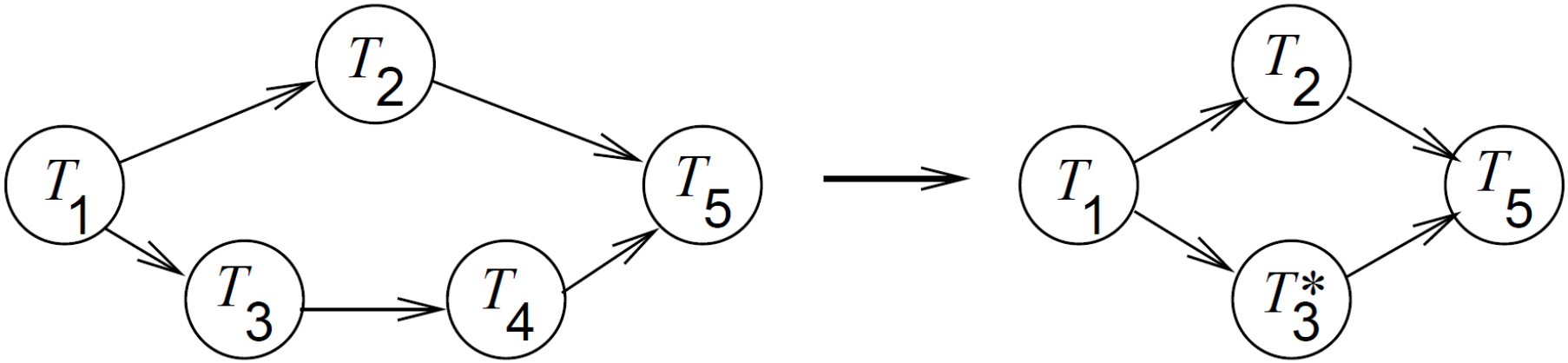
Book section 7.1

Granularity: size of tasks (e.g. in instructions)

Readable specifications and efficient implementations can possibly require different task structures.

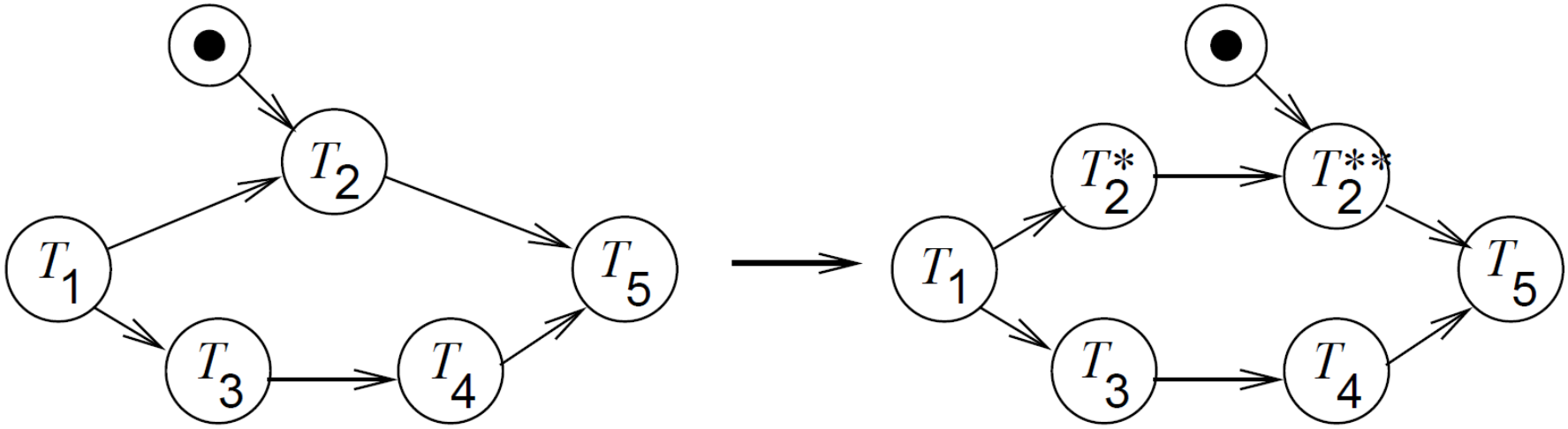
☞ Granularity changes

Merging of tasks



- Reduced overhead of context switches
- More global optimization of machine code
- Reduced overhead for inter-process/task communication

Splitting of tasks



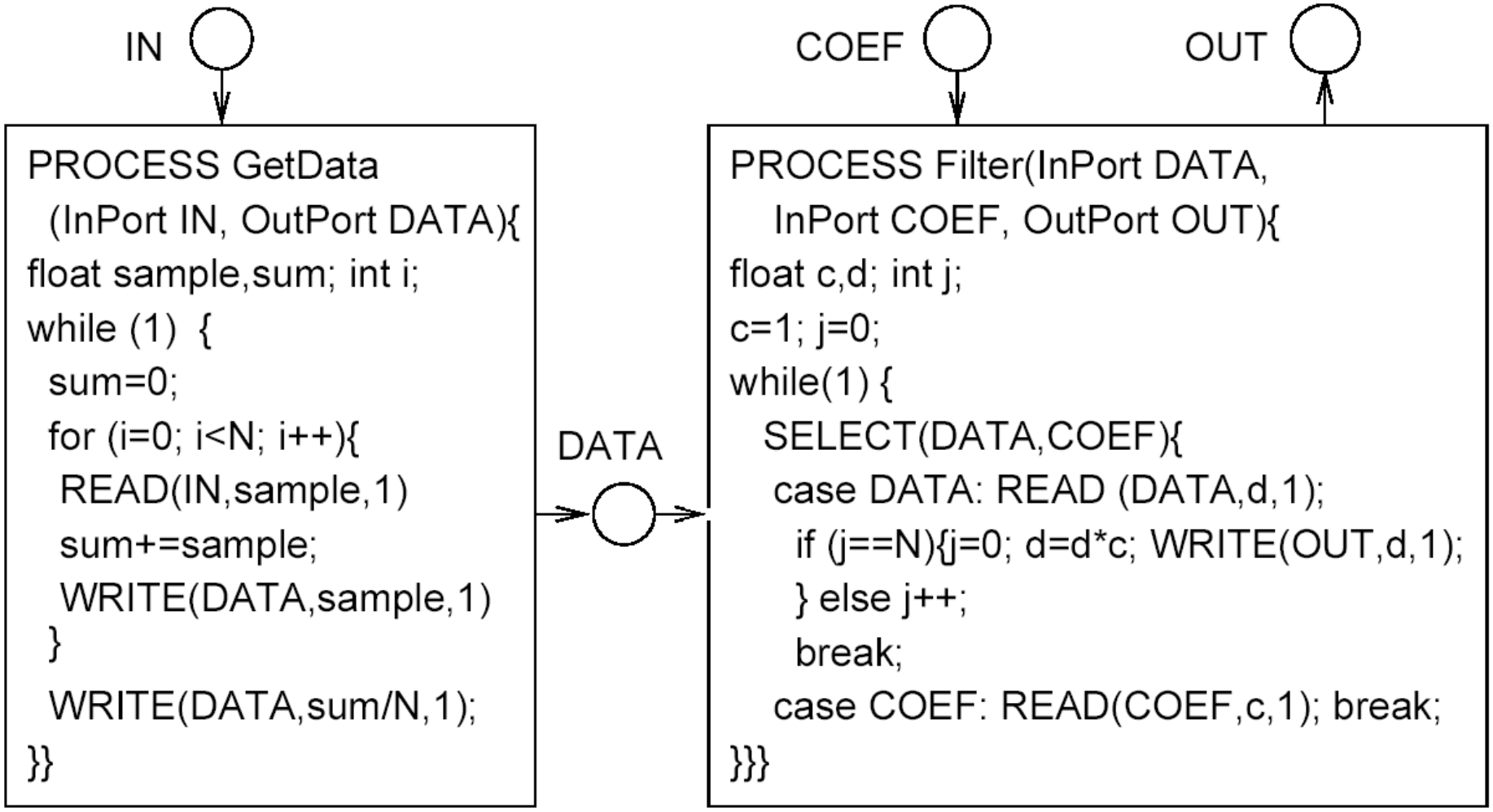
- No blocking of resources while waiting for input
- More flexibility for scheduling, possibly improved result

Merging and splitting of tasks

The most appropriate task graph granularity depends upon the context ☞ merging and splitting may be required.

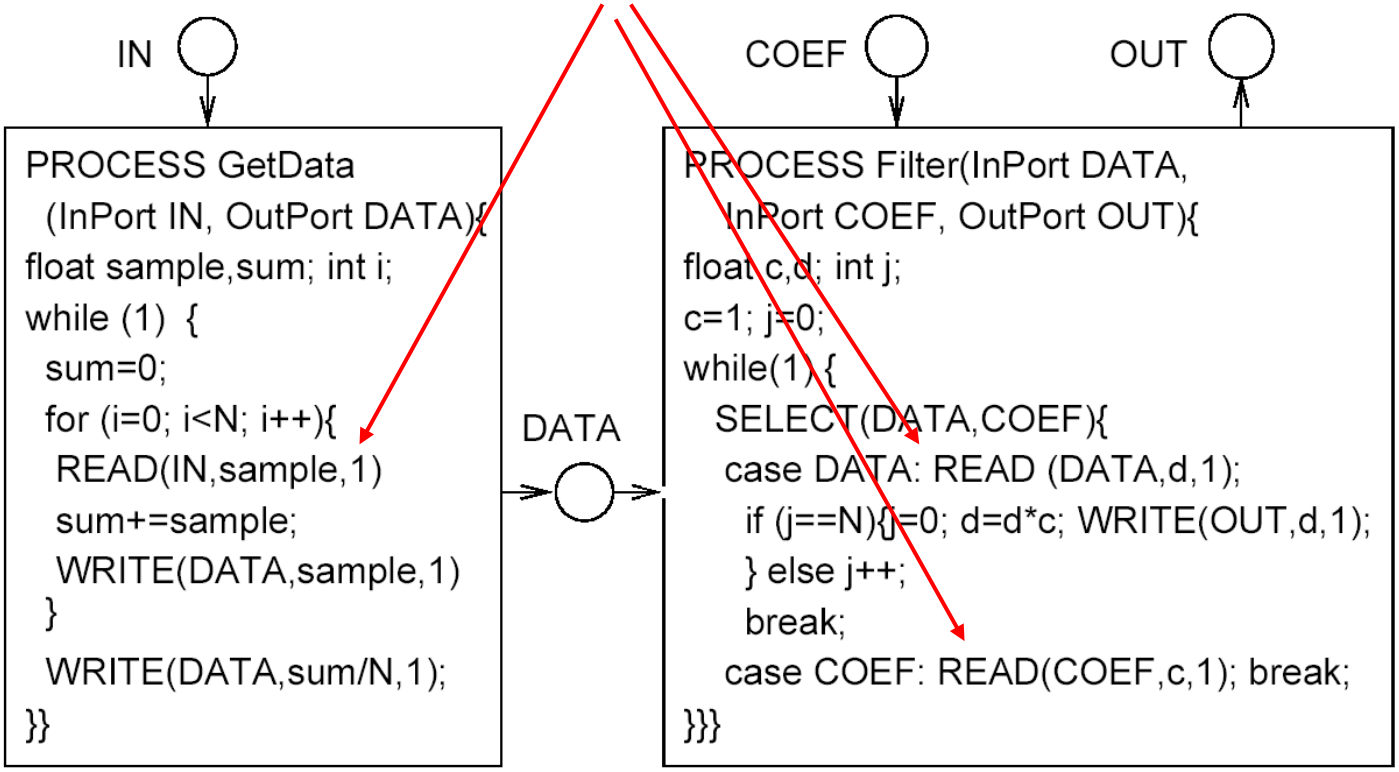
Merging and splitting of tasks should be done automatically, depending upon the context.

Automated rewriting of the task system
- Example -



Attributes of a system that needs rewriting

Tasks blocking after they have already started running

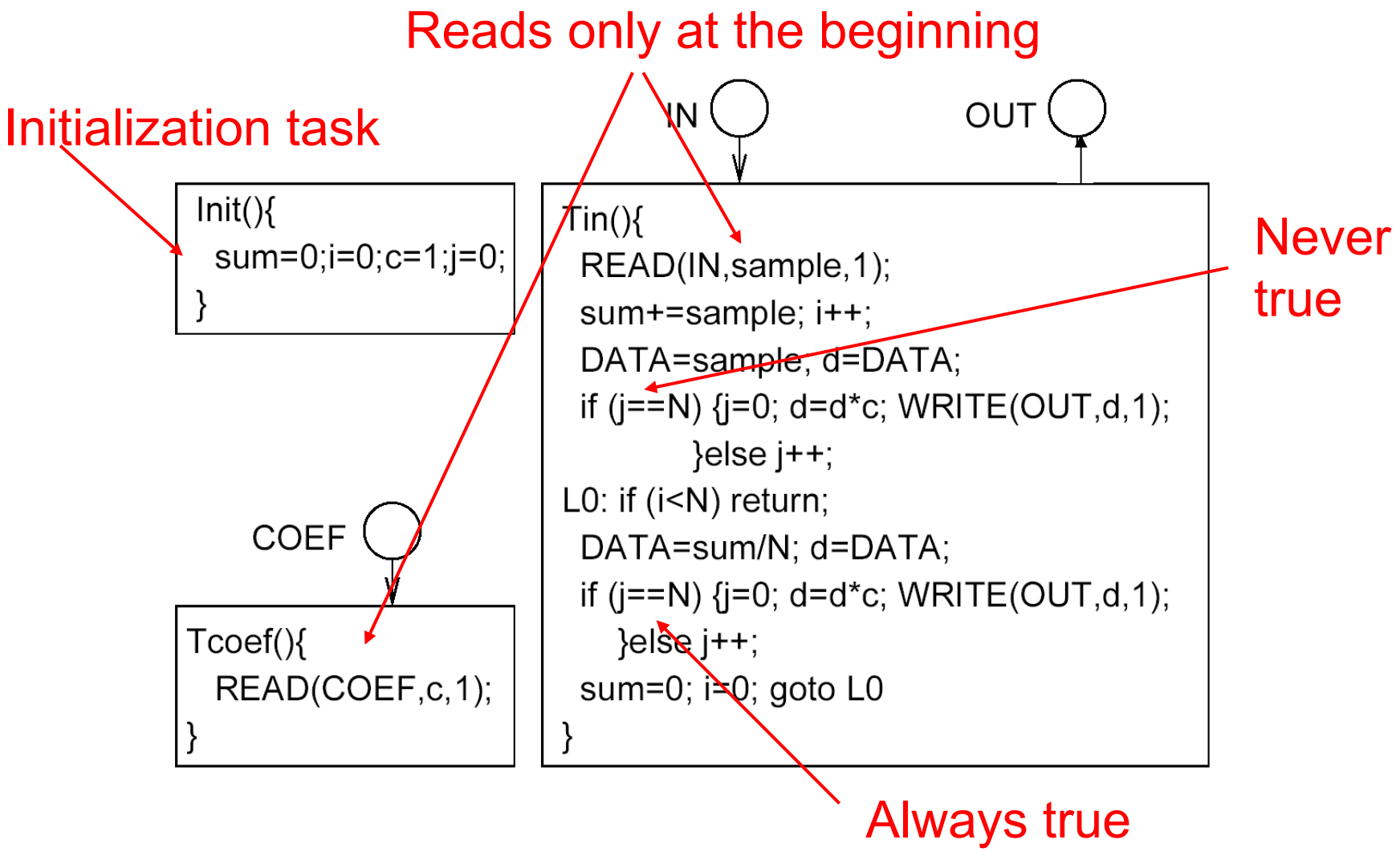


Work by Cortadella et al.

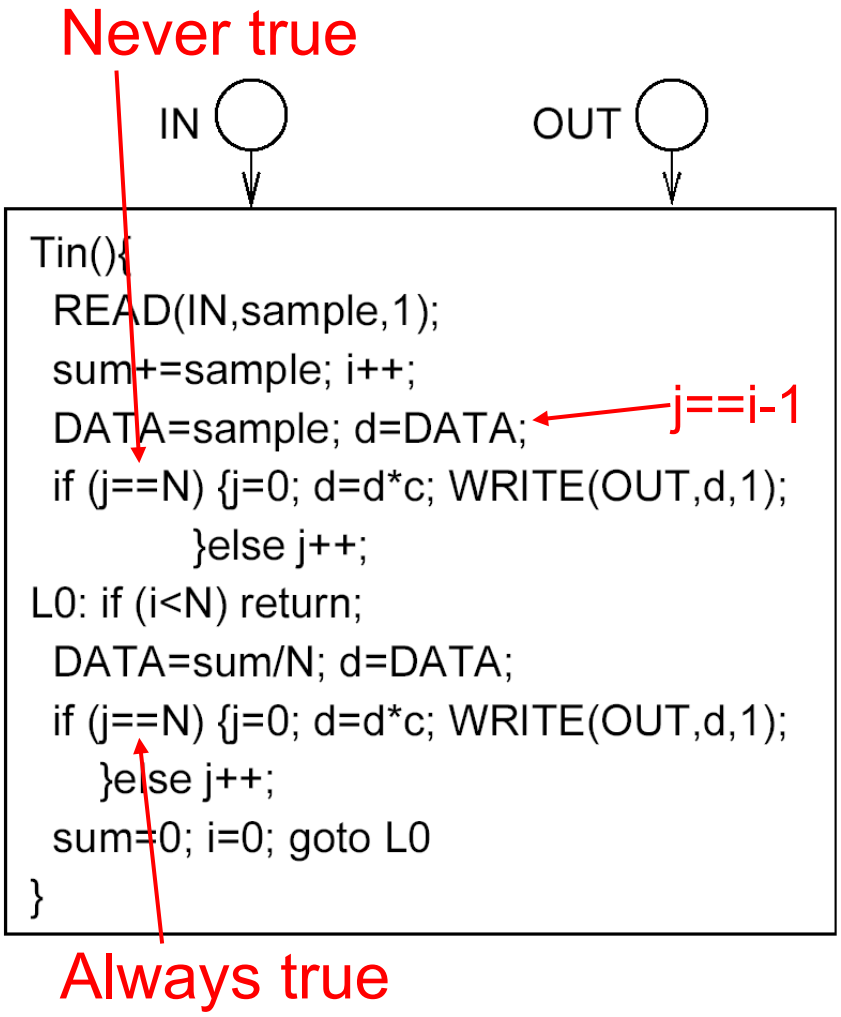
1. Transform each of the tasks into a Petri net,
2. Generate one global Petri net from the nets of the tasks,
3. Partition global net into “sequences of transitions”
4. Generate one task from each such sequence

Mature, commercial approach not yet available

Result, as published by Cortadella



Optimized version of Tin

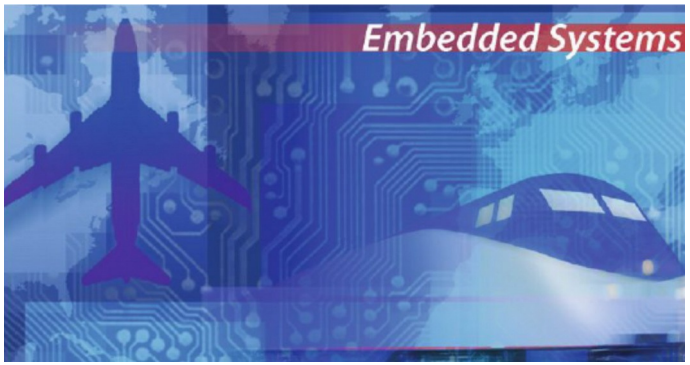


j ☞ i

```
Tin () {
    READ (IN, sample, 1);
    sum += sample; i++;
    DATA = sample; d = DATA;
L0: if (i < N) return;
    DATA = sum/N; d = DATA;
    d = d*c; WRITE(OUT,d,1);
    sum = 0; i = 0;
    return;
}
```

High-level software transformations

Peter Marwedel
TU Dortmund,
Informatik 12



High-level optimizations

Book section 7.2

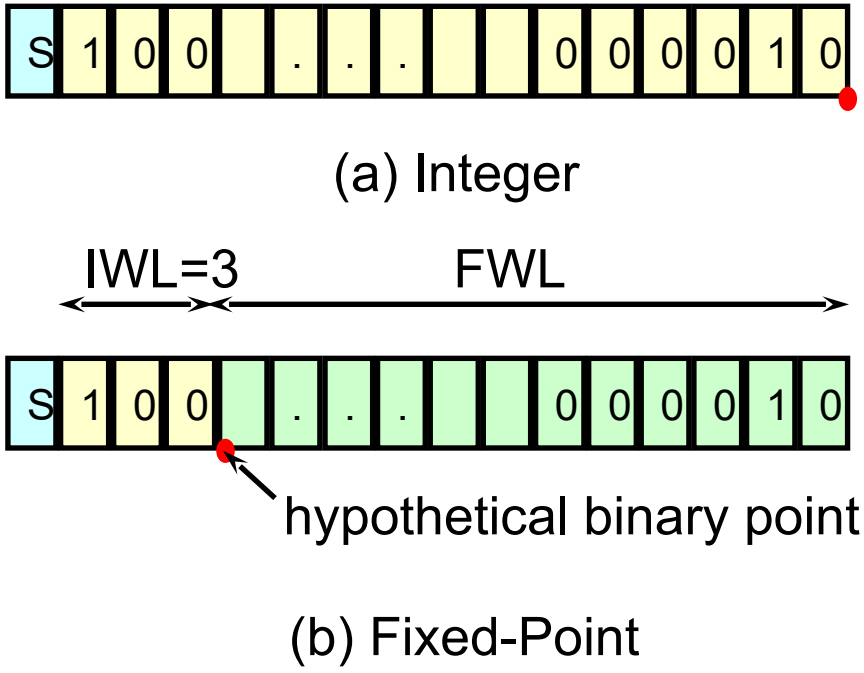
- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding

Fixed-Point Data Format

- Floating-Point vs. Fixed-Point

- *exponent*, mantissa
- Floating-Point
  - automatic computation and update of each exponent at run-time
- Fixed-Point
  - implicit exponent
  - determined off-line

- Integer vs. Fixed-Point



Floating-point to fixed point conversion

PROs:

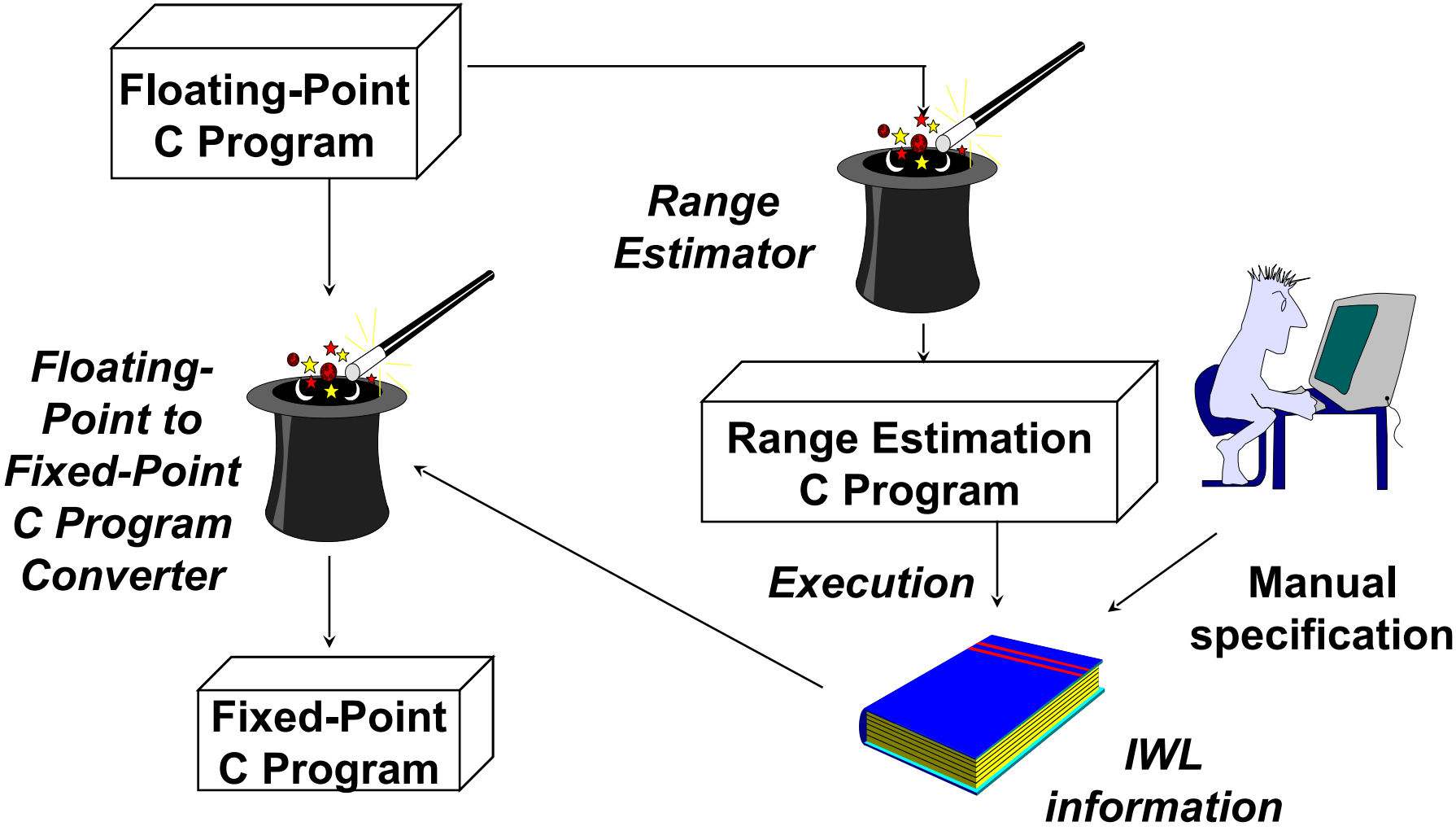
- Lower cost
- Faster
- Lower power consumption
- Sufficient SQNR, *if properly scaled*
- Suitable for portable applications

CONs:

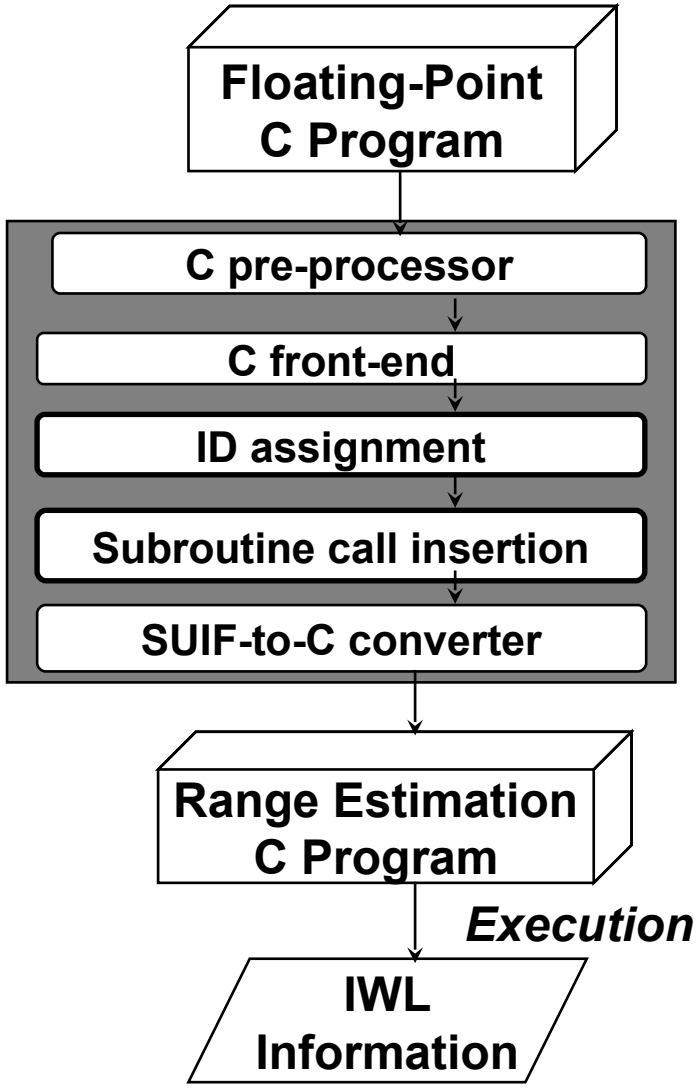
- Decreased dynamic range
- Finite word-length effect, *unless properly scaled*
  - Overflow and excessive quantization noise
- Extra programming effort

© *Ki-Il Kum, et al. (*Seoul National University)*:* A Floating-point To Fixed-point C Converter For Fixed-point Digital Signal Processors, 2nd SUIF Workshop, 1996

Development Procedure



Range Estimator



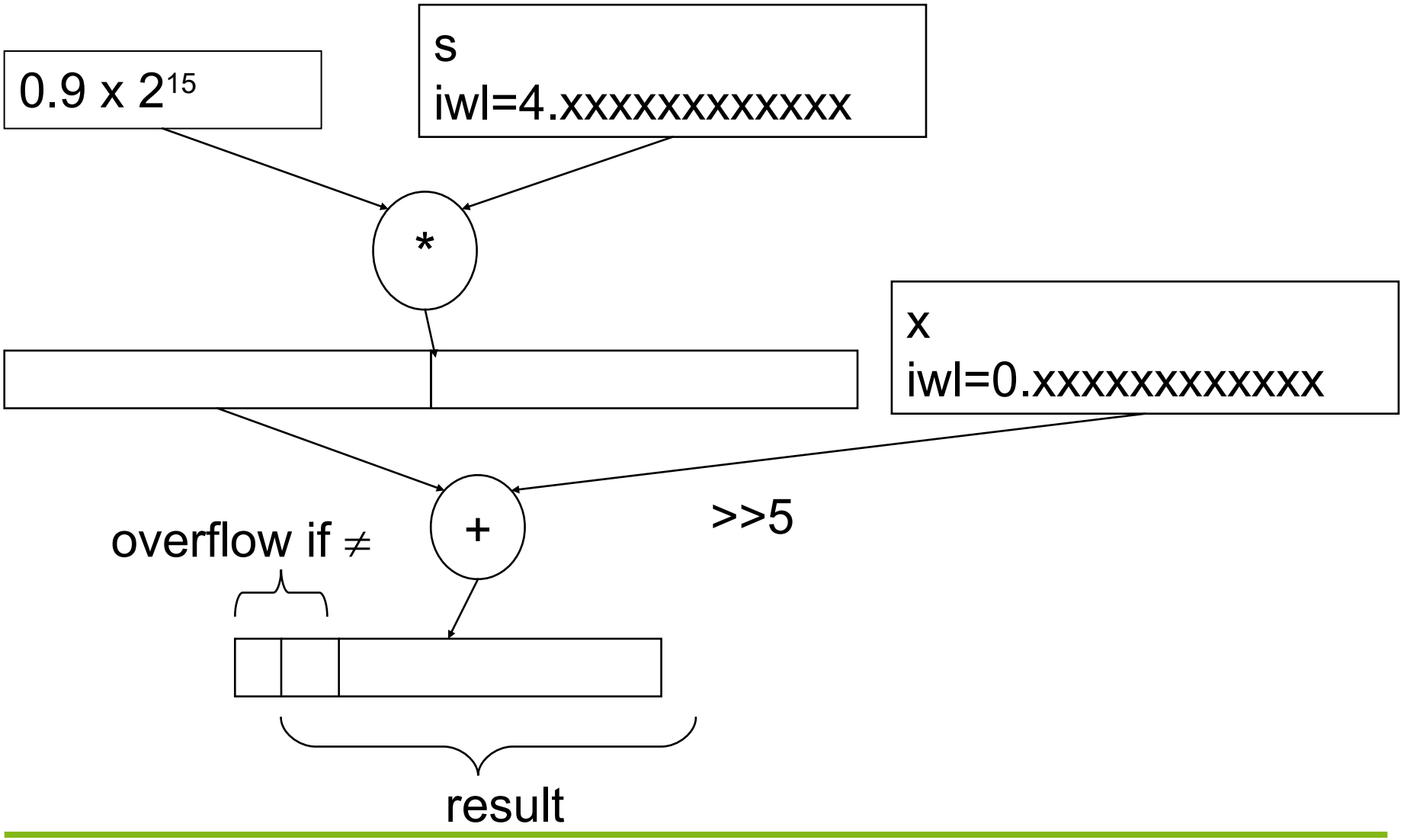
Range Estimation C Program

```
float iir1(float x)
{
    static float s = 0;
    float y;

    y = 0.9 * s + x;
    range(y, 0);
    s = y;
    range(s, 1);

    return y;
}
```

Operations in fixed point program



Floating-Point to Fixed-Point Program Converter

Fixed-Point C Program

```
int iir1(int x)
{
    static int s = 0;
    int y;
    y=sll(mulh(29491,s)+ (x>>5),1);
    s = y;
    return y;
}
```

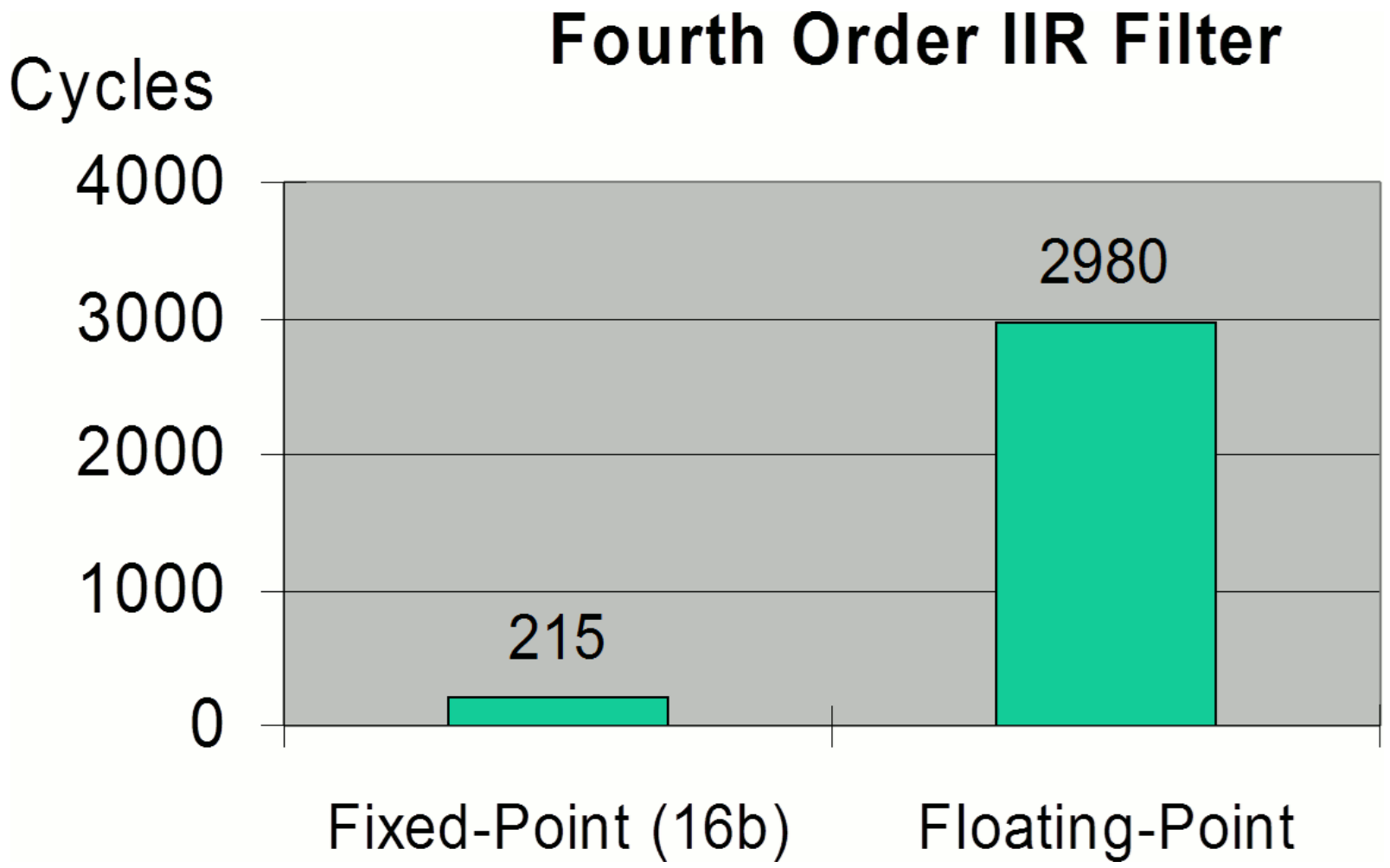
mulh

- to access the upper half of the multiplied result
- target dependent implementation

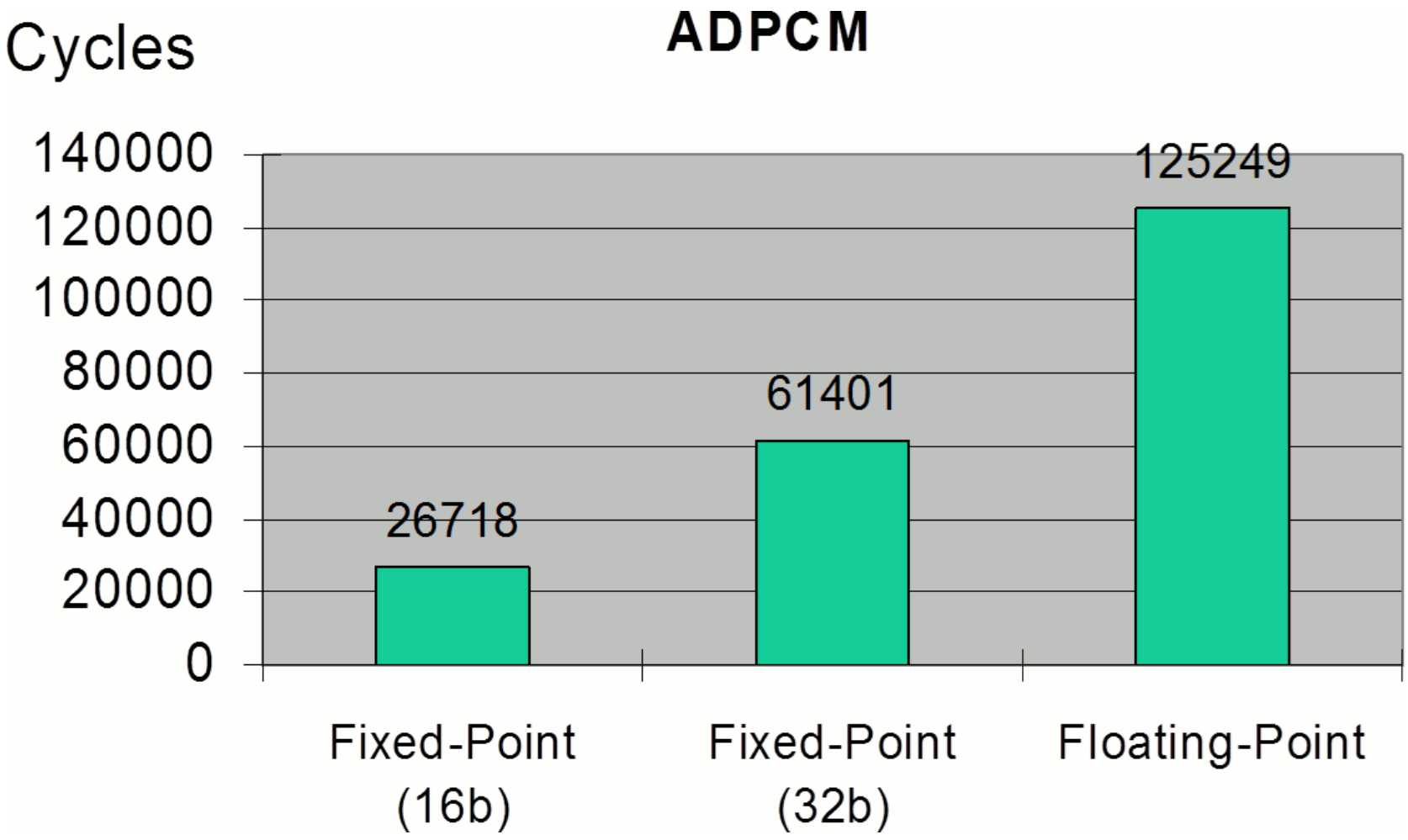
sll

- to remove 2nd sign bit
- opt. overflow check

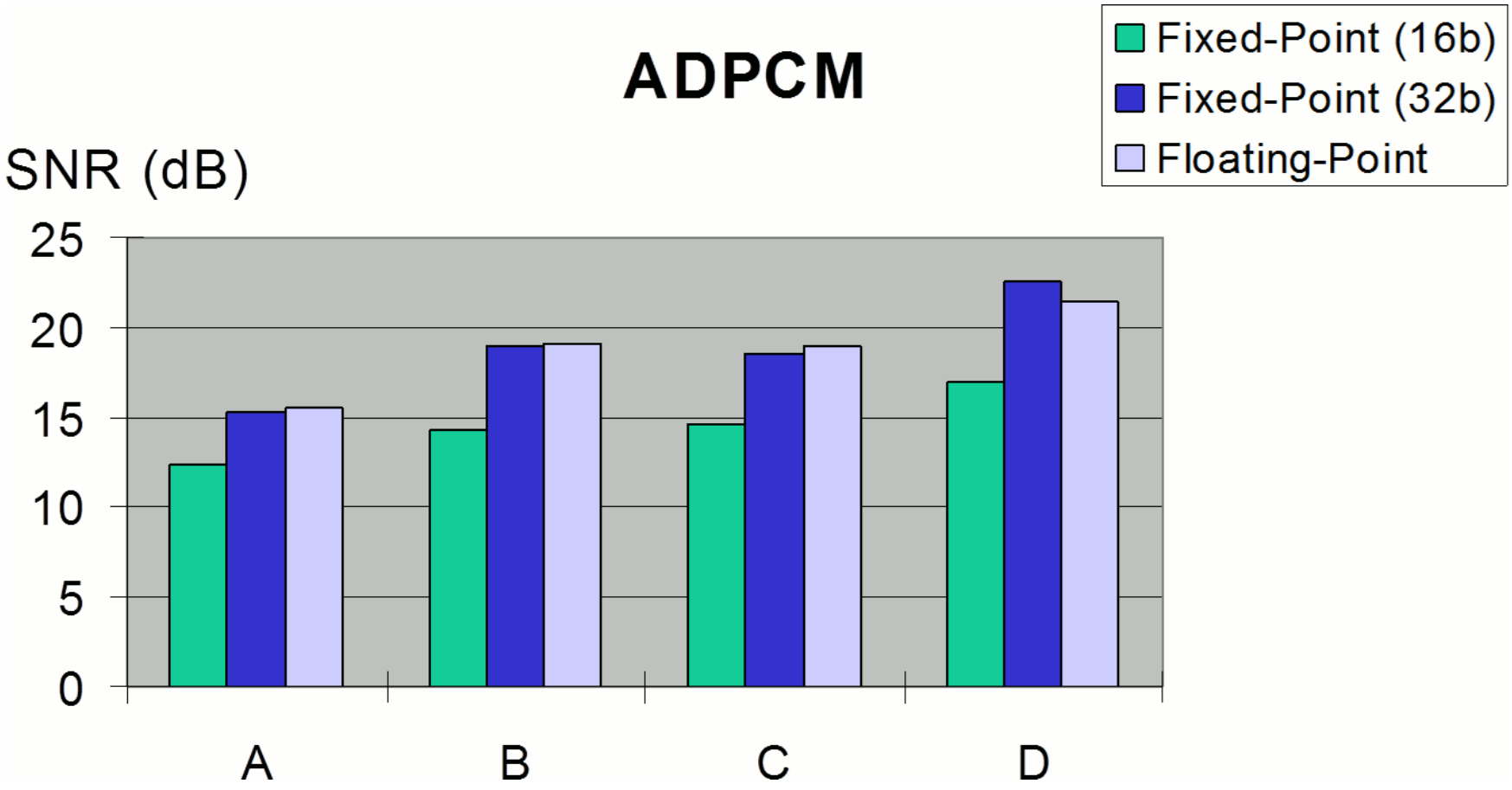
Performance Comparison - Machine Cycles - (1)



Performance Comparison - Machine Cycles - (2)



Performance Comparison - SNR -



High-level optimizations

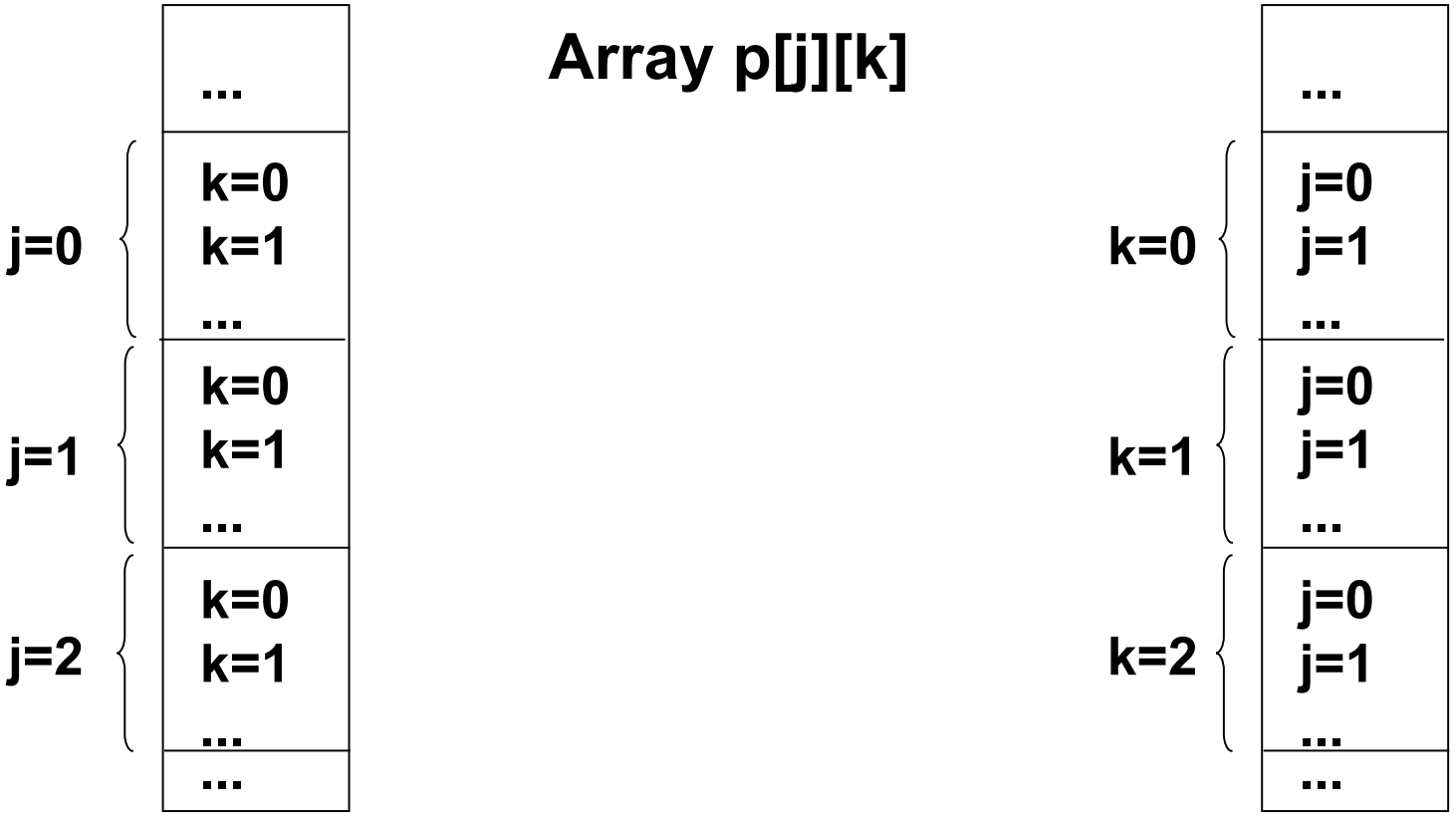
Book section 7.2

- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding

Impact of memory allocation on efficiency

Row major order
(C)

Column major order
(FORTRAN)



Best performance if innermost loop corresponds to rightmost array index

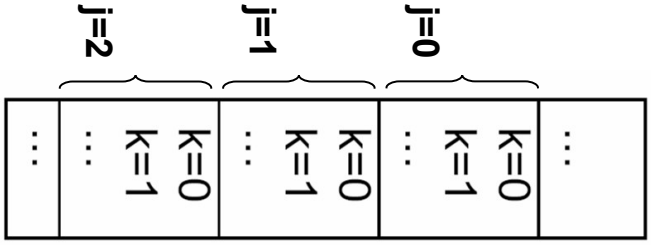
Two loops, assuming row major order (C):

```
for (k=0; k<=m; k++)
  for (j=0; j<=n; j++) )
    p[j][k] = ...
```

```
for (j=0; j<=n; j++)
  for (k=0; k<=m; k++)
    p[j][k] = ...
```

Same behavior for homogeneous memory access, but:

For row major order



↑ Poor cache behavior Good cache behavior ↑

☞ memory architecture dependent optimization

☞ Program transformation “Loop interchange”

Example:

☞ Improved locality

```
... #define iter 400000
int a[20][20][20];
void computeijk() {int i,j,k;
  for (i = 0; i < 20; i++) {
    for (j = 0; j < 20; j++) {
      for (k = 0; k < 20; k++) {
        a[i][j][k] += a[i][j][k];}}}}
void computeikj() {int i,j,k;
  for (i = 0; i < 20; i++) {
    for (j = 0; j < 20; j++) {
      for (k = 0; k < 20; k++) {
        a[i][k][j] += a[i][k][j];}}}} ...
start=time(&start); for(z=0; z<iter; z++) computeijk();
  end=time(&end);
printf("ijk=%16.9f\n", 1.0*difftime(end,start)); ...
```

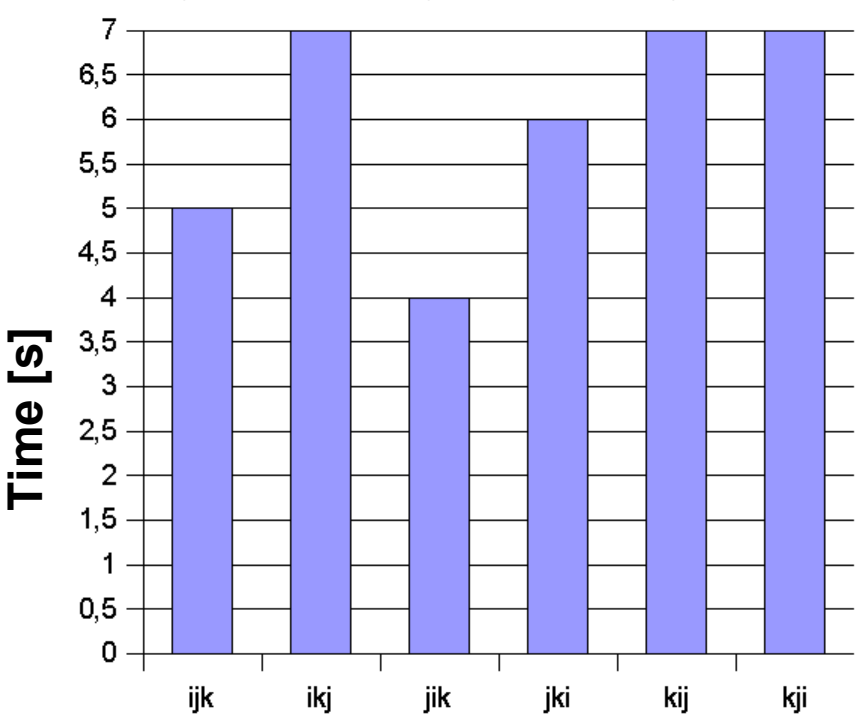
(SUIF interchanges array indexes instead of loops)

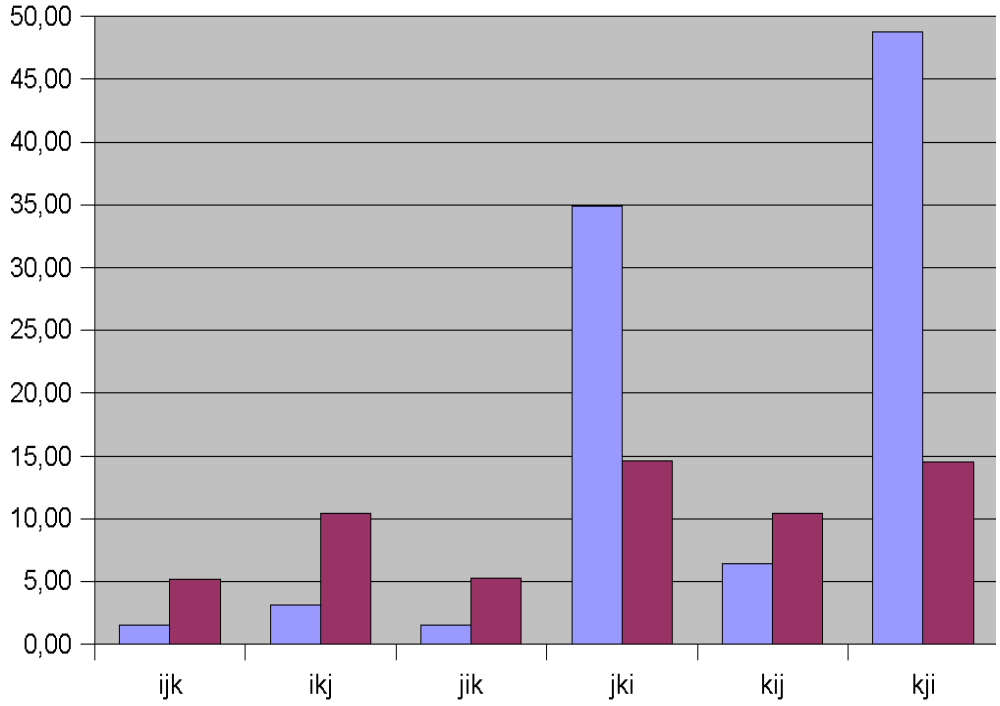
Results:
strong influence of the memory architecture

Loop structure: i j k

Dramatic impact of locality

Processor	Ti C6xx	Sun SPARC	Intel Pentium
reduction to [%]	~ 57%	35%	3.2 %



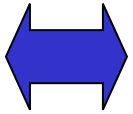


Not always the same impact ..

Till Buchwald, Diploma thesis, Univ. Dortmund, Informatik 12, 12/2004

Transformations
“Loop fusion” (merging), “loop fission”

```
for(j=0; j<=n; j++)
  p[j]= ... ;
for (j=0; j<=n; j++) ,
  p[j]= p[j] + ...
```



```
for (j=0; j<=n; j++)
  { p[j]= ... ;
    p[j]= p[j] + ...}
```

Loops small enough to allow zero overhead Loops.

Better locality for access to p.
Better chances for parallel execution.

Which of the two versions is best?

Architecture-aware compiler should select best version.

Example: simple loops

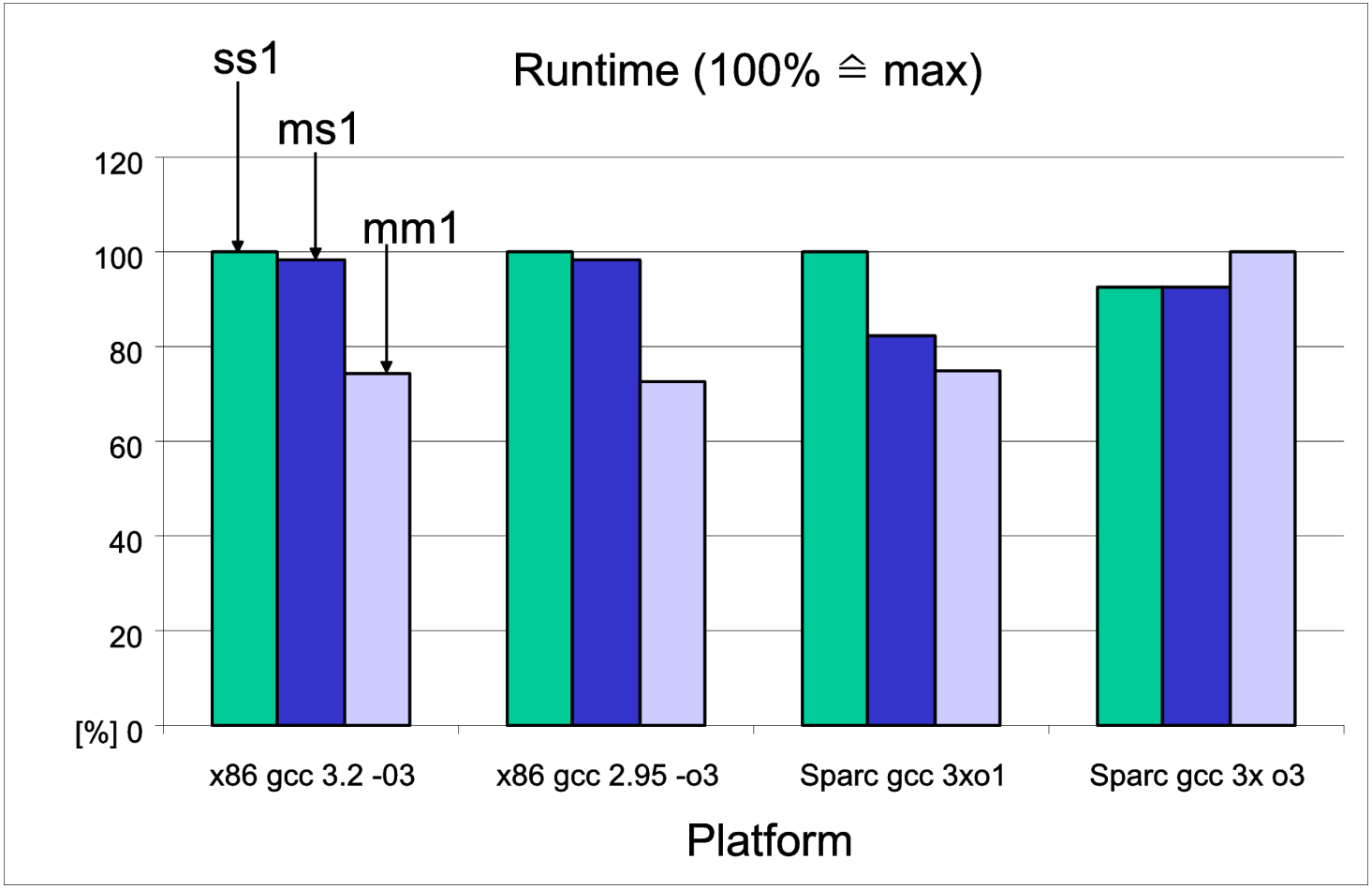
```
#define size 30
#define iter 40000
int   a[size][size];
float b[size][size];
```

```
void ms1() {int i,j;
 for (i=0;i< size;i++){
  for (j=0;j<size;j++){
   a[i][j]+=17;     }
  for (j=0;j<size;j++){
   b[i][j]-=13; }}}
```

```
void ss1() {int i,j;
 for (i=0;i<size;i++){
  for (j=0;j<size;j++){
   a[i][j]+= 17;}}
 for(i=0;i<size;i++){
  for (j=0;j<size;j++){
   b[i][j]-=13;}}}
```

```
void mm1() {int i,j;
 for(i=0;i<size;i++){
  for(j=0;j<size;j++){
   a[i][j] += 17;
   b[i][j] -= 13;}}}
```

Results: simple loops



☞

Merged loops superior; except Sparc with –o3

Loop unrolling

```
for (j=0; j<=n; j++)
    p[j]= ... ;
```

```
for (j=0; j<=n; j+=2)
    {p[j]= ... ; p[j+1]= ...}
```

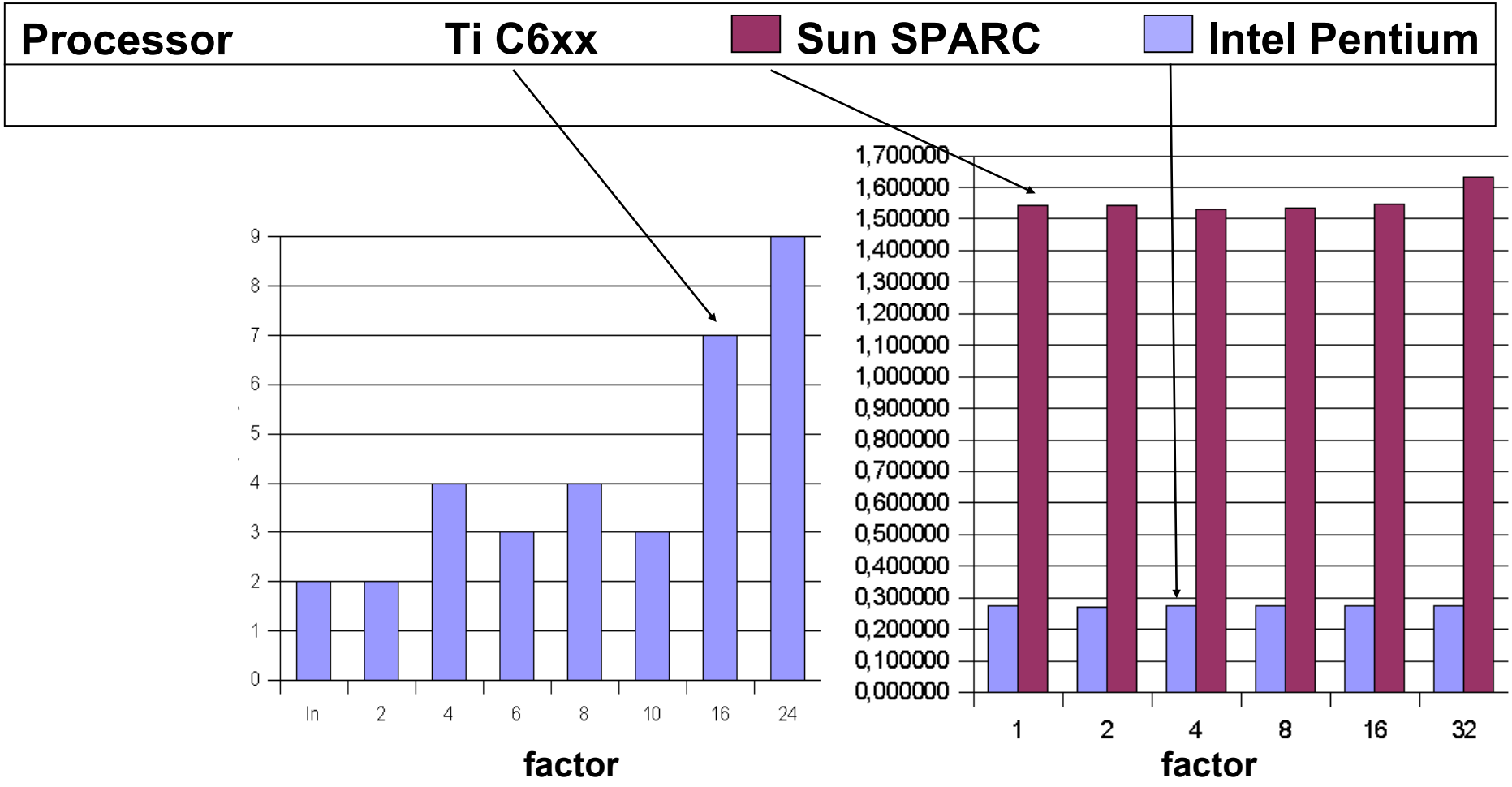
- Factor = 2
- Better locality for access to p.
- Less branches per execution of the loop. More opportunities for optimizations.
- Tradeoff between code size and improvement.
- Extreme case: completely unrolled loop (no branch)

Example: matrixmult

```
#define s 30
#define iter 4000
int a[s][s],b[s]
[s],c[s][s];
void compute(){int
i,j,k;
 for(i=0;i<s;i++){
  for(j=0;j<s;j++){
   for(k=0;k<s;k++){
    c[i][k]+=
    a[i][j]*b[j][k];
}}}}
```

```
extern void compute2()
 {int i, j, k;
  for (i = 0; i < 30; i++) {
   for (j = 0; j < 30; j++)  {
    for (k = 0; k <= 28; k += 2)
     {{int *suif_tmp;
     suif_tmp = &c[i][k];
     *suif_tmp=
     *suif_tmp+a[i][j]*b[j][k];}
     {int *suif_tmp;
     suif_tmp=&c[i][k+1];
     *suif_tmp=*suif_tmp
            +a[i][j]*b[j][k+1];
  }}}}
 return;}
```

Results



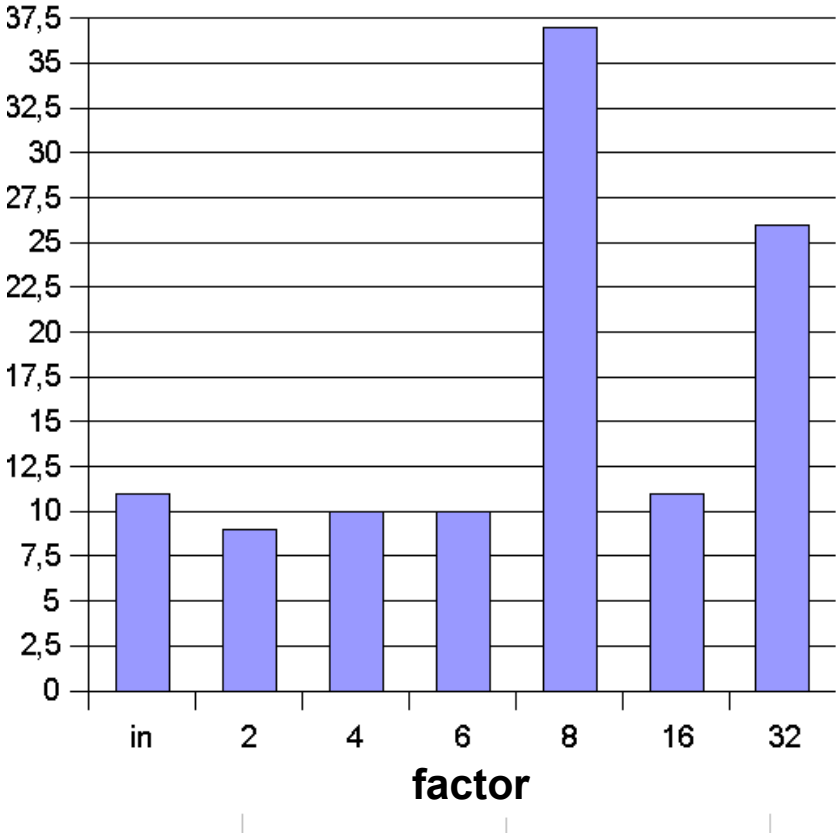
Benefits quite small; penalties may be large

Till Buchwald, Diploma thesis,
Univ. Dortmund, Informatik 12, 12/2004

Results: benefits for loop dependences

Processor	Ti C6xx
reduction to [%]	

```
#define s 50
#define iter 150000
int a[s][s], b[s][s];
void compute() {
 int i,k;
  for (i = 0; i < s; i++) {
   for (k = 1; k < s; k++) {
    a[i][k] = b[i][k];
    b[i][k] = a[i][k-1];
}}}
```



Small benefits

Till Buchwald, Diploma thesis, Univ. Dortmund, Informatik 12, 12/2004

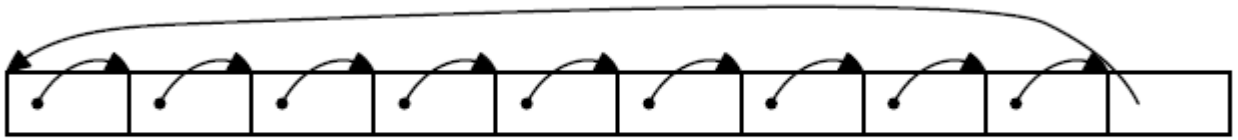
High-level optimizations

Book section 7.2

- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding

Impact of caches on execution times?

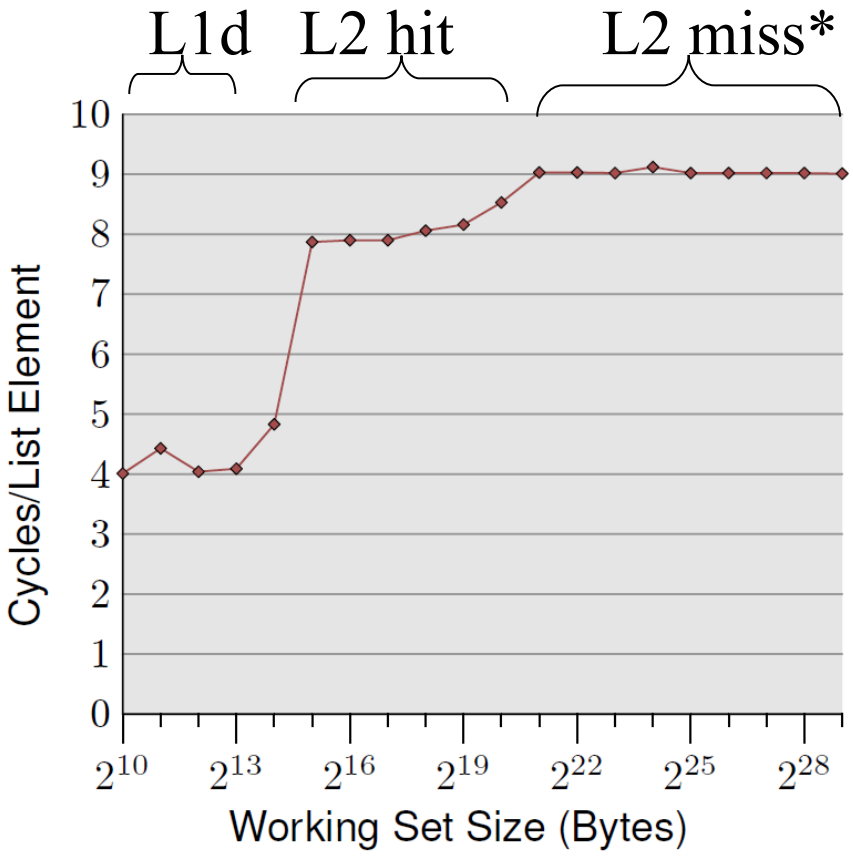
- Execution time for traversal of linked list, stored in an array, each entry comprising NPAD*8 Bytes



- Pentium P4
  - 16 KB L1 data cache, 4 cycles/access
  - 1 MB L2 cache, 14 cycles/access
  - Main memory, 200 cycles/access

U. Drepper: *What every programmer should know about memory**, 2007, http://www.akkadia.org/drepper/cpumemory.pdf; Dank an Prof. Teubner (LS6) für Hinweis auf diese Quelle
* In Anlehnung an das Papier „David Goldberg, *What every programmer should know about floating point arithmetic, ACM Computing Surveys*, 1991 (auch für diesen Kurs benutzt).

Cycles/access as a function of the size of the list



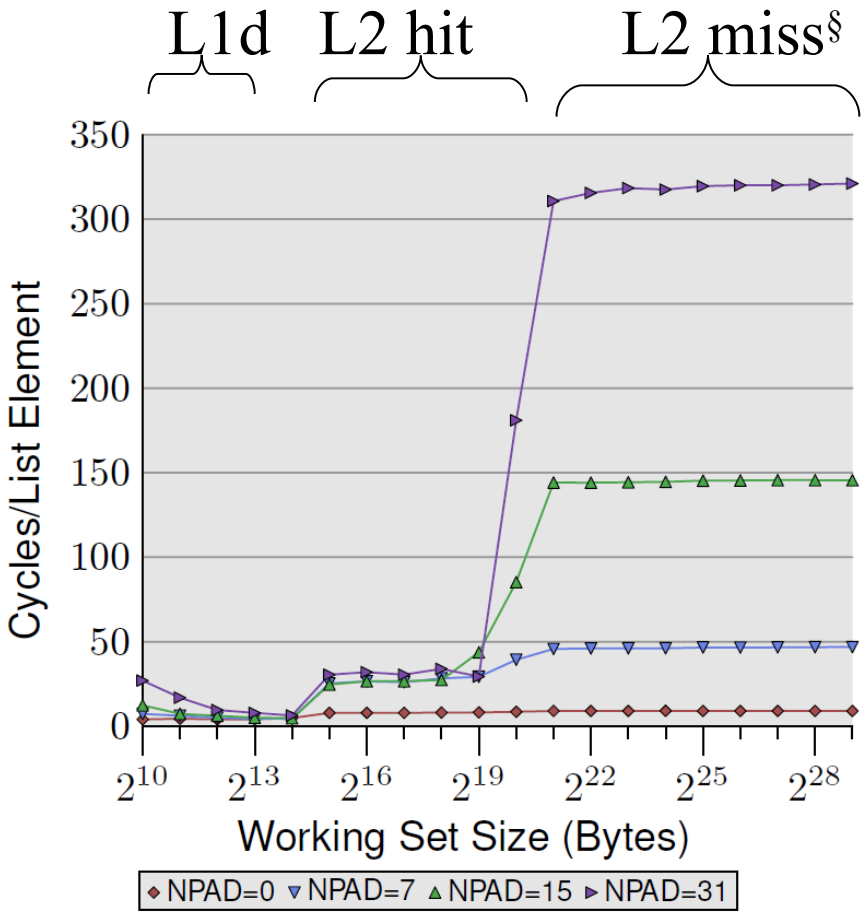


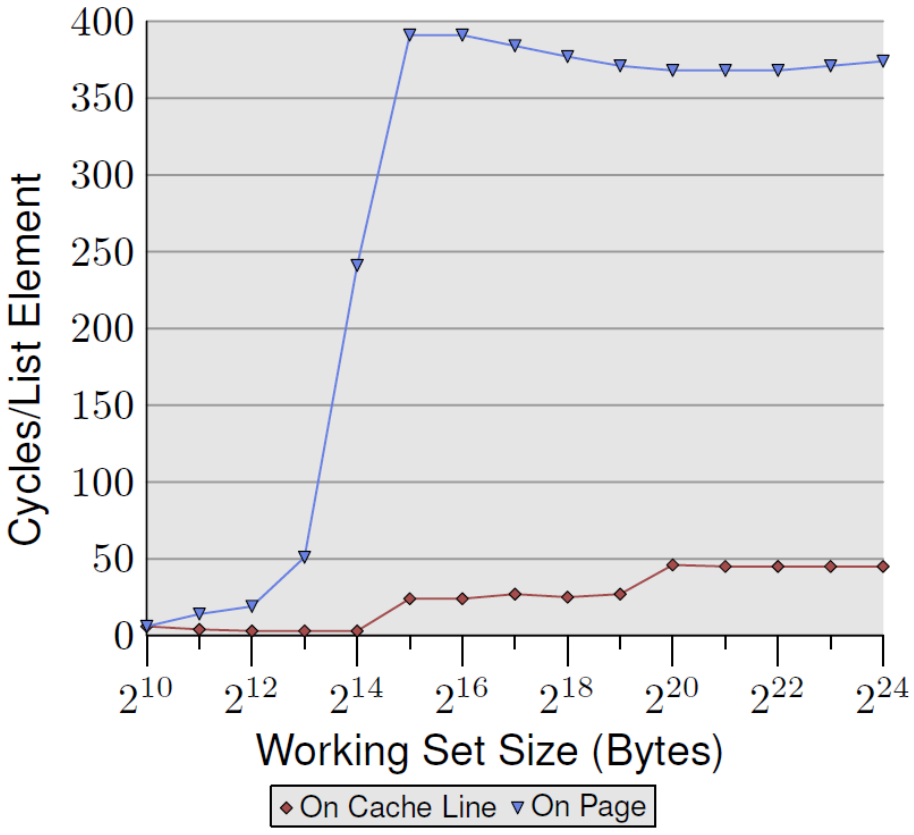
Figure 3.10: Sequential Read Access, NPAD=0

* prefetching succeeds

§ prefetching fails

Impact of TLB misses and larger caches

Elements on different pages; run time increase when exceeding the size of the TLB



Larger caches are shifting the steps to the right

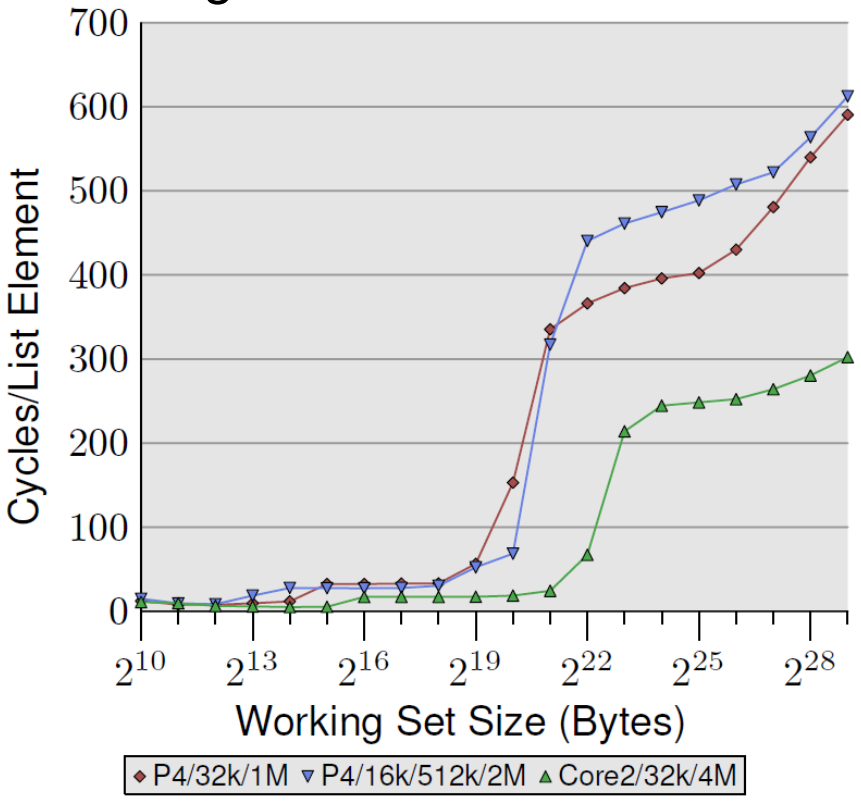
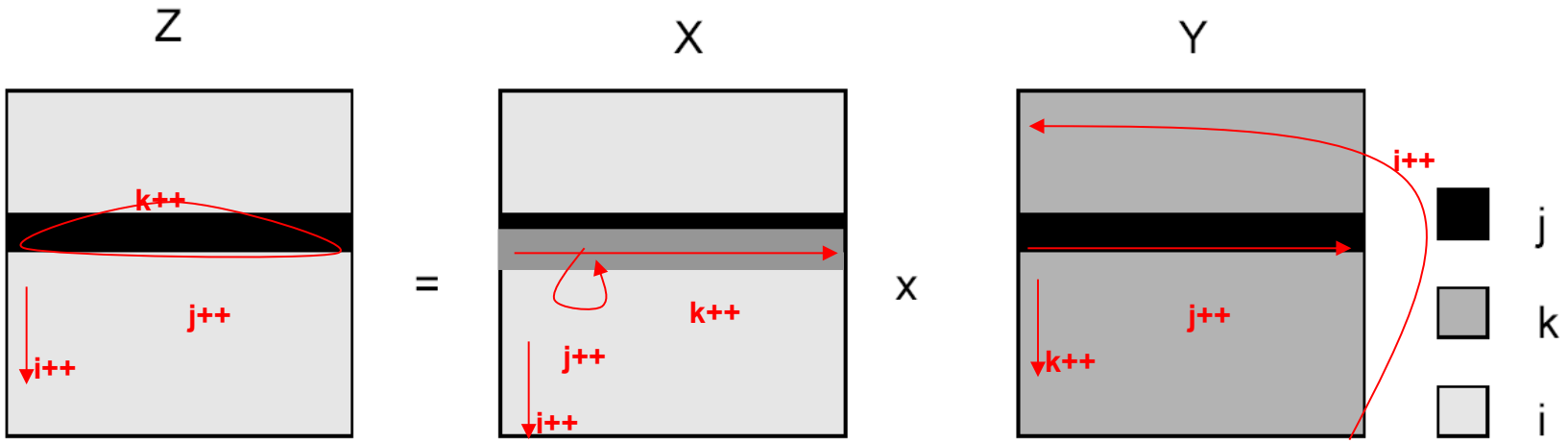


Figure 3.14: Advantage of Larger L2/L3 Caches

Program transformation
Loop tiling/loop blocking: - Original version -

```
for (i=1; i<=N; i++)
    for(k=1; k<=N; k++){
        r=X[i,k]; /* to be allocated to a register*/
        for (j=1; j<=N; j++)
            Z[i,j] += r* Y[k,j]
} % Never reusing information in the cache for Y and Z if N
    is large or cache is small (2 N³ references for Z).
```



Loop tiling/loop blocking
- tiled version -

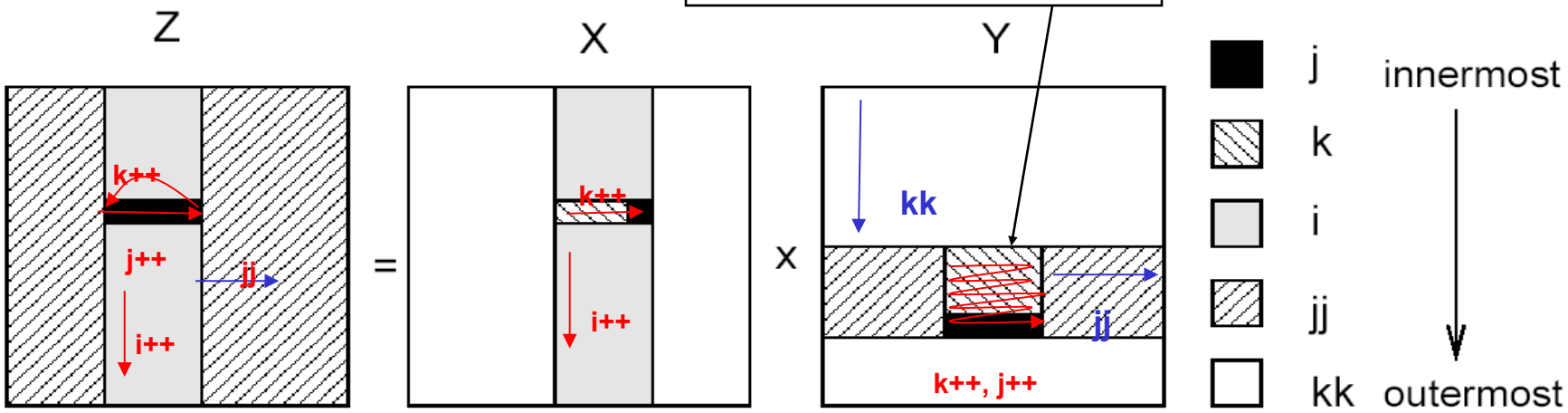
```
for (kk=1; kk<= N; kk+=B)
  for (jj=1; jj<= N; jj+=B)
    for (i=1; i<= N; i++)
      for (k=kk; k<= min(kk+B-1,N); k++){
        r=X[i][k]; /* to be allocated to a register*/
        for (j=jj; j<= min(jj+B-1, N); j++)
          Z[i][j] += r* Y[k][j]
        }
```

Reuse factor of B for Z, N for Y

$O(N^3/B)$ accesses to main memory

Compiler should select best option

Same elements for next iteration of i

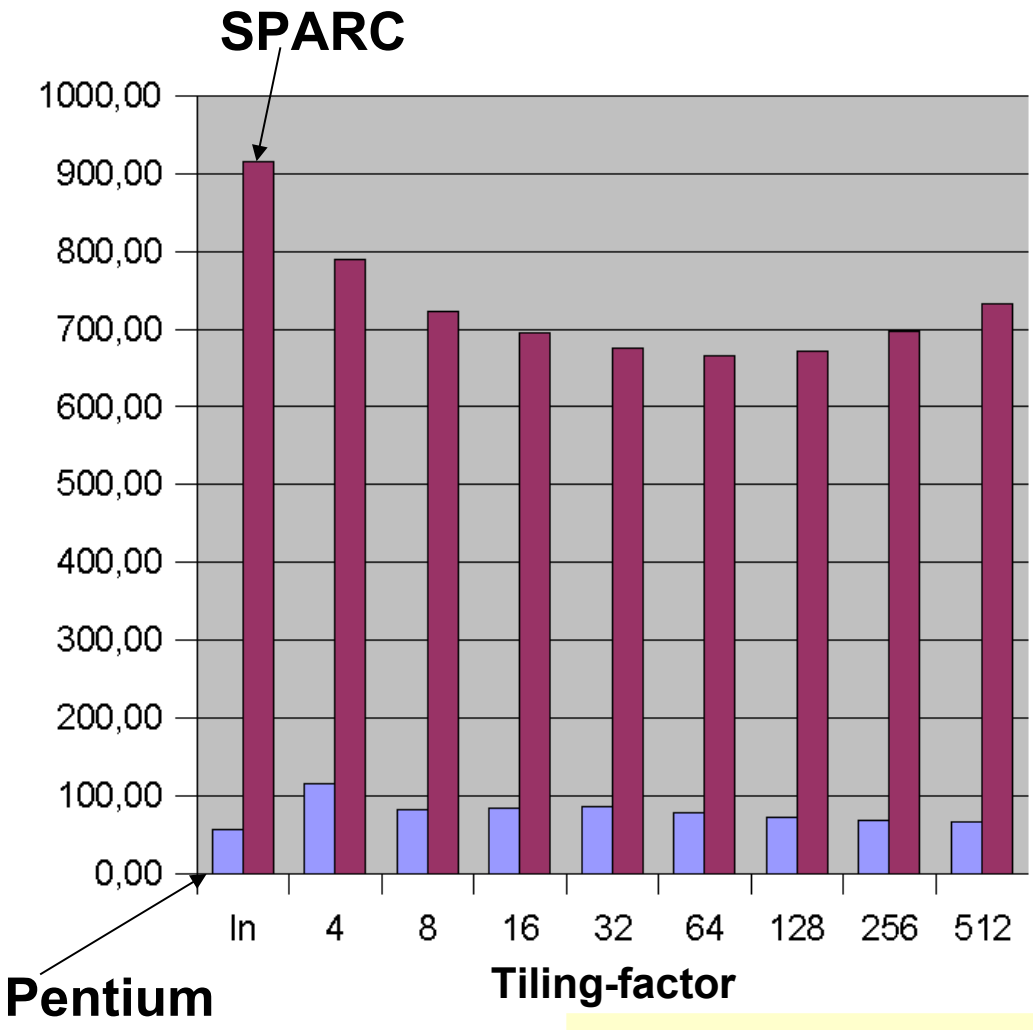


Monica Lam: The Cache Performance and Optimization of Blocked Algorithms, ASPLOS, 1991

Example

In practice, results are disappointing.

One of the few cases where an improvement was achieved.
Source: similar to matrix mult.



Till Buchwald, Diploma thesis,
Univ. Dortmund, Informatik 12, 12/2004

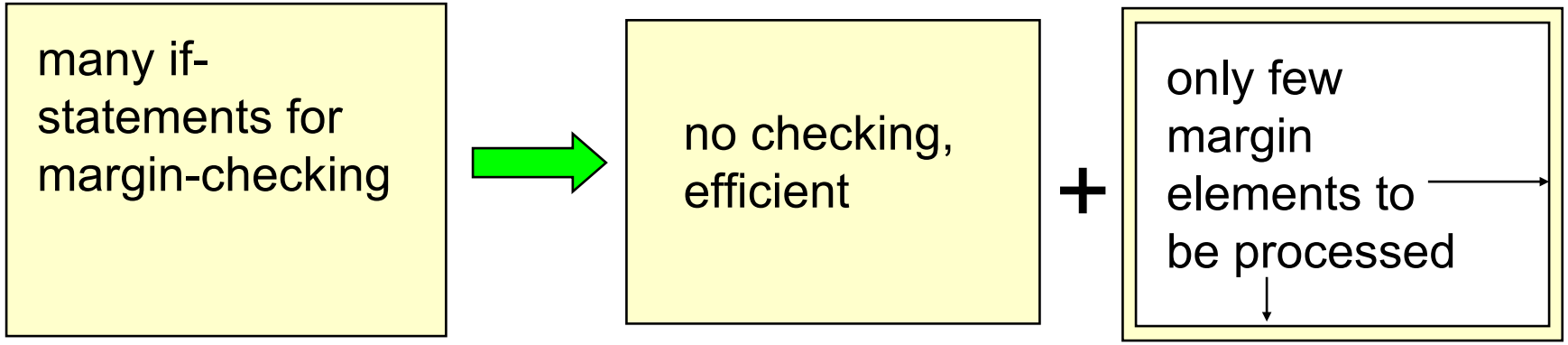
High-level optimizations

Book section 7.2

- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding

Transformation “Loop nest splitting”

Example: Separation of margin handling



Loop nest splitting at University of Dortmund

Loop nest from MPEG-4 full search motion estimation

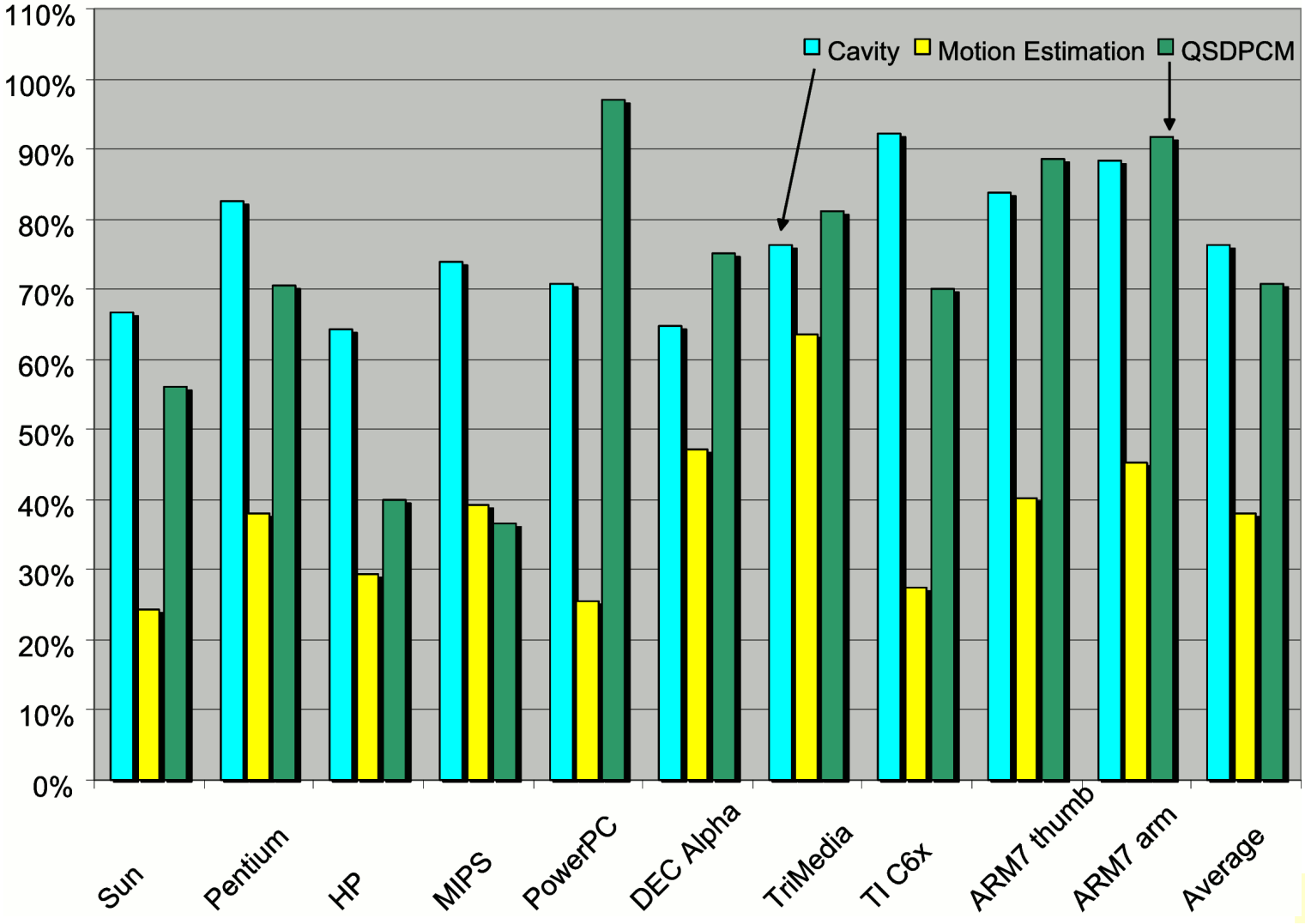
```
for (z=0; z<20; z++)
 for (x=0; x<36; x++) {x1=4*x;
  for (y=0; y<49; y++) {y1=4*y;
   for (k=0; k<9; k++) {x2=x1+k-4;
    for (l=0; l<9; ) {y2=y1+l-4;
     for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
      for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
       if (x3<0 || 35<x3||y3<0||48<y3)
        then_block_1; else else_block_1;
       if (x4<0|| 35<x4||y4<0||48<y4)
        then_block_2; else else_block_2;
}}}}}}
```

analysis of polyhedral domains, selection with genetic algorithm

```
for (z=0; z<20; z++)
 for (x=0; x<36; x++) {x1=4*x;
  for (y=0; y<49; y++)
if (x>=10||y>=14)
  for (; y<49; y++)
    for (k=0; k<9; k++)
     for (l=0; l<9;l++ )
      for (i=0; i<4; i++)
       for (j=0; j<4;j++) {
        then_block_1; then_block_2}
else {y1=4*y;
  for (k=0; k<9; k++) {x2=x1+k-4;
    for (l=0; l<9; ) {y2=y1+l-4;
     for (i=0; i<4; i++) {x3=x1+i; x4=x2+i;
      for (j=0; j<4;j++) {y3=y1+j; y4=y2+j;
       if (0 || 35<x3 ||0 || 48<y3)
        then-block-1; else else-block-1;
       if (x4<0|| 35<x4||y4<0||48<y4)
        then_block_2; else else_block_2;
}}}}}}
```

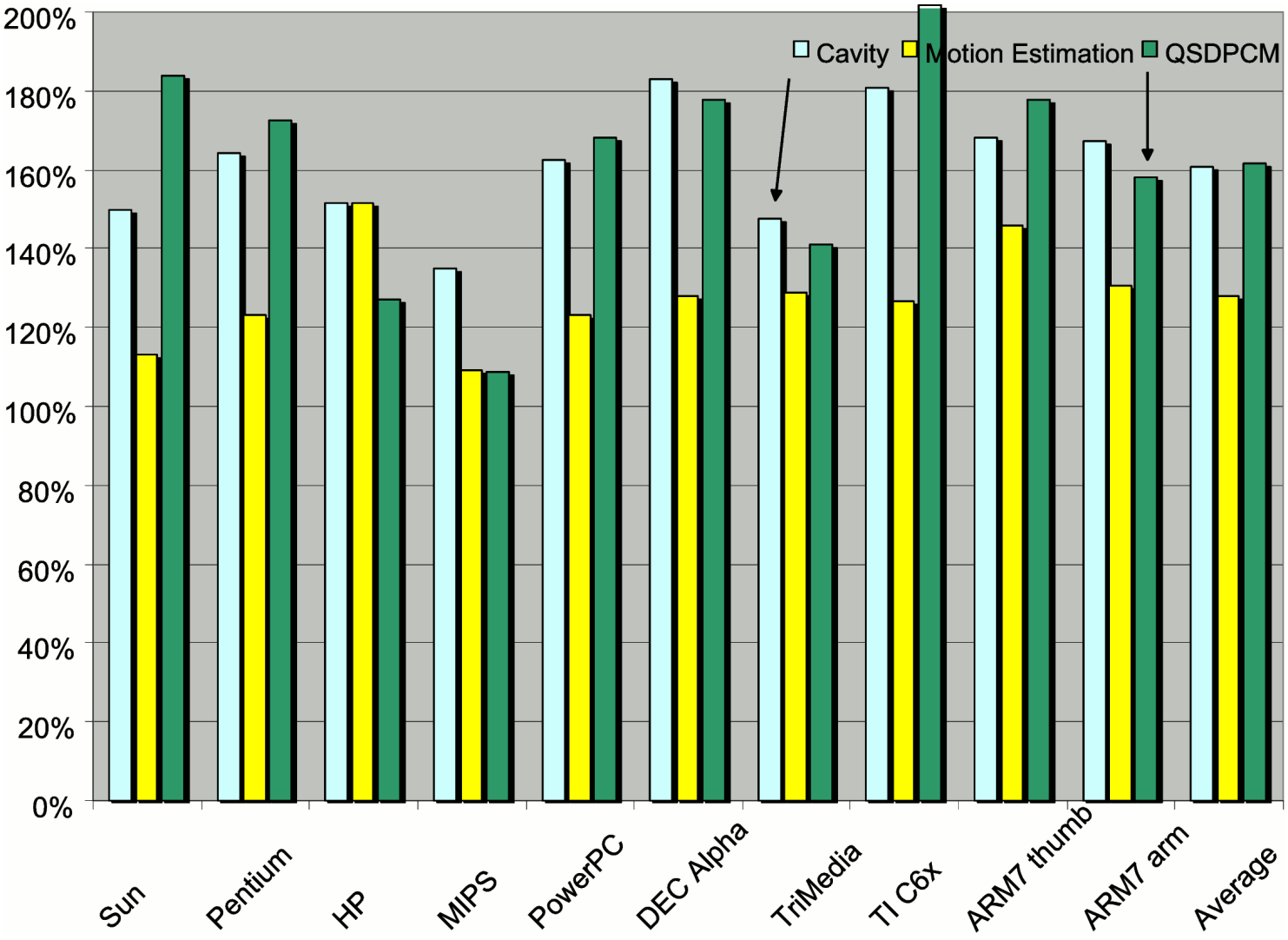
H. Falk et al., Inf 12, UniDo, 2002

Results for loop nest splitting
- Execution times -



H. Falk et al., 2002

Results for loop nest splitting
- Code sizes -



H. Falk et al., 2002

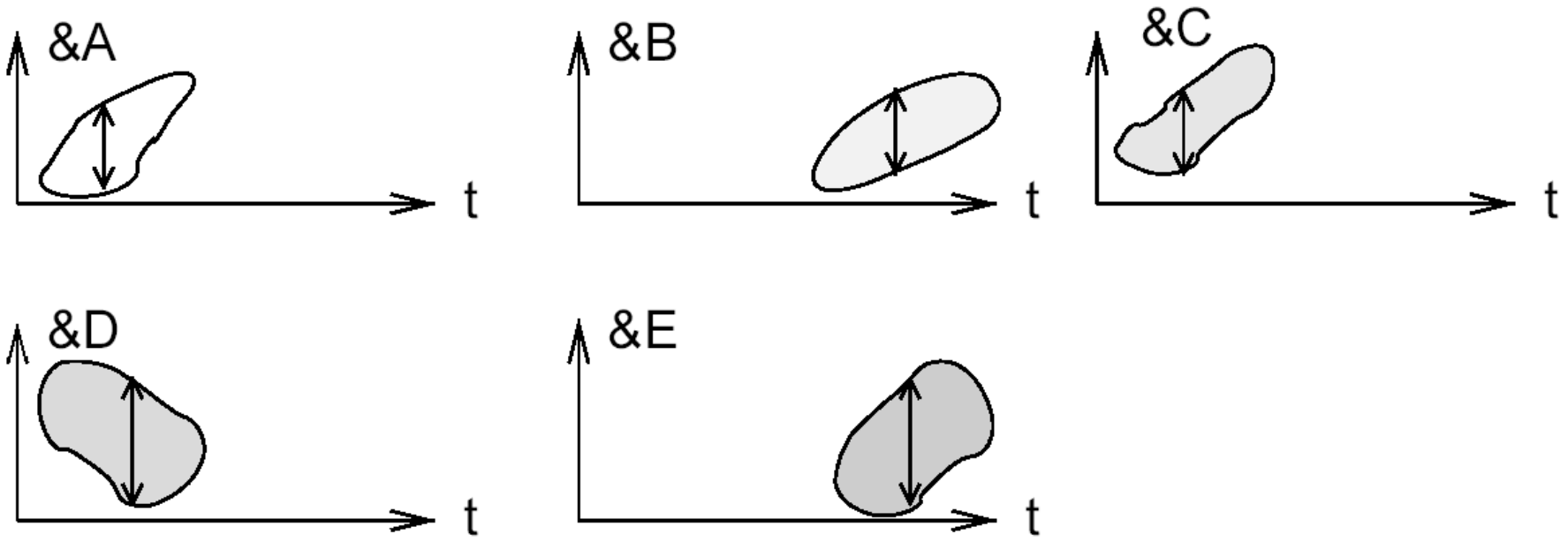
High-level optimizations

Book section 7.2

- Floating-point to fixed point conversion
- Simple loop transformations
- Loop tiling/blocking
- Loop (nest) splitting
- Array folding

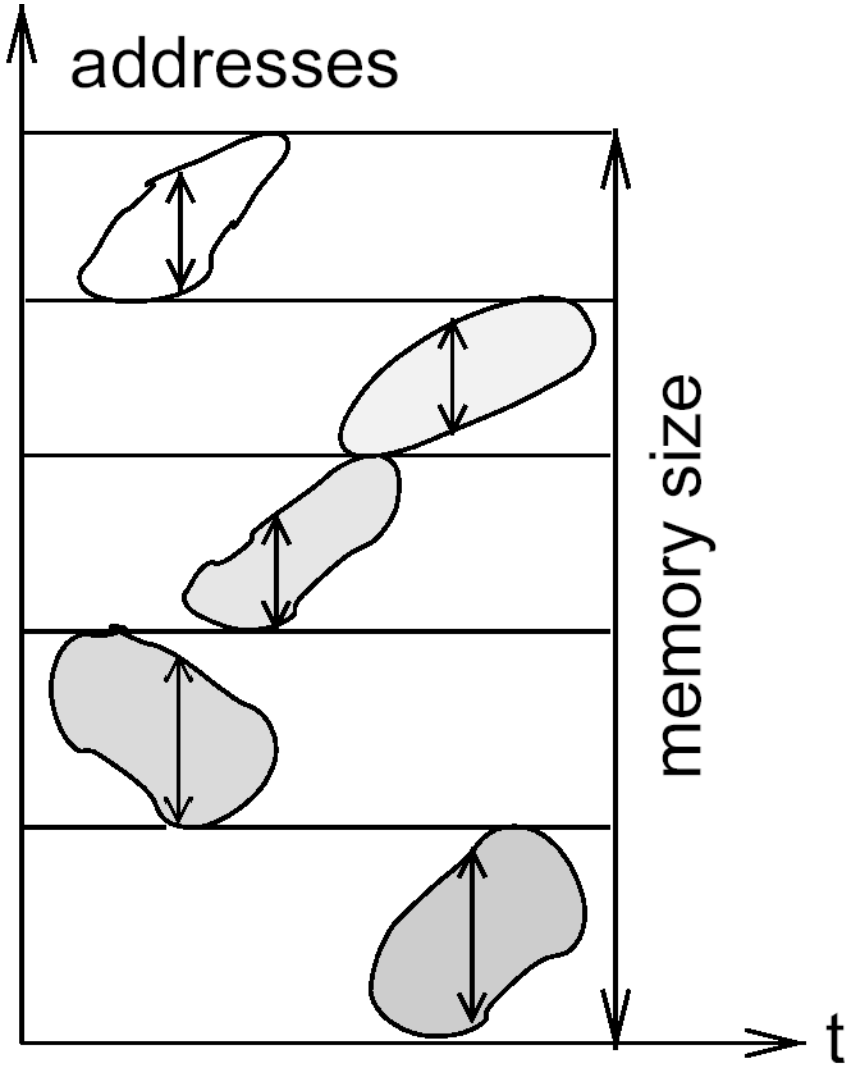
Array folding

Initial arrays

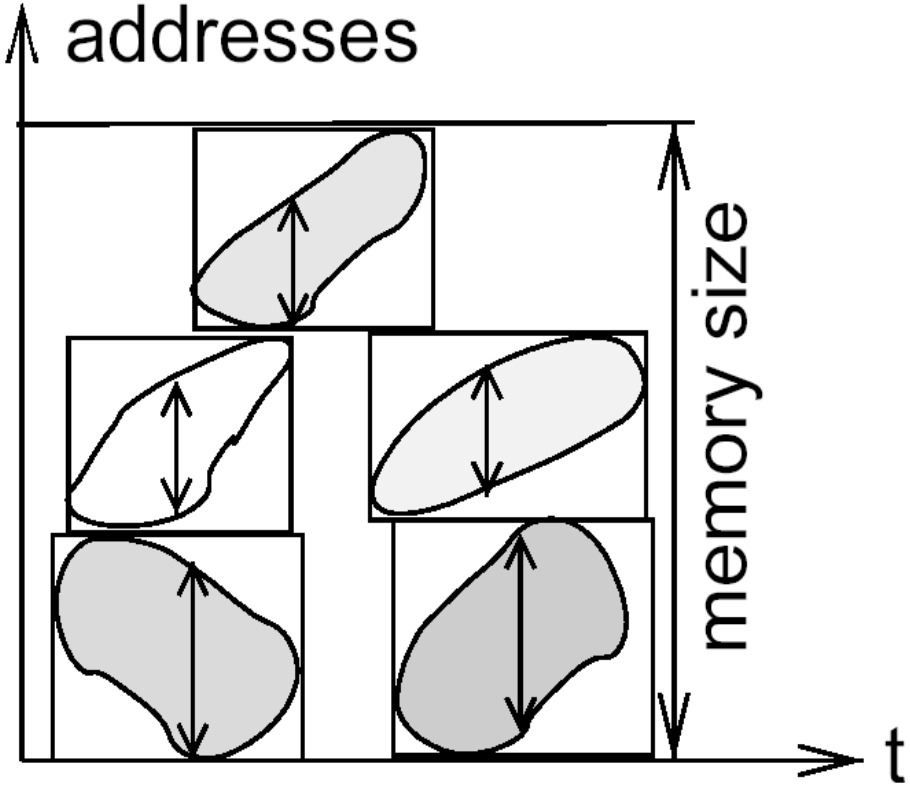


Array folding

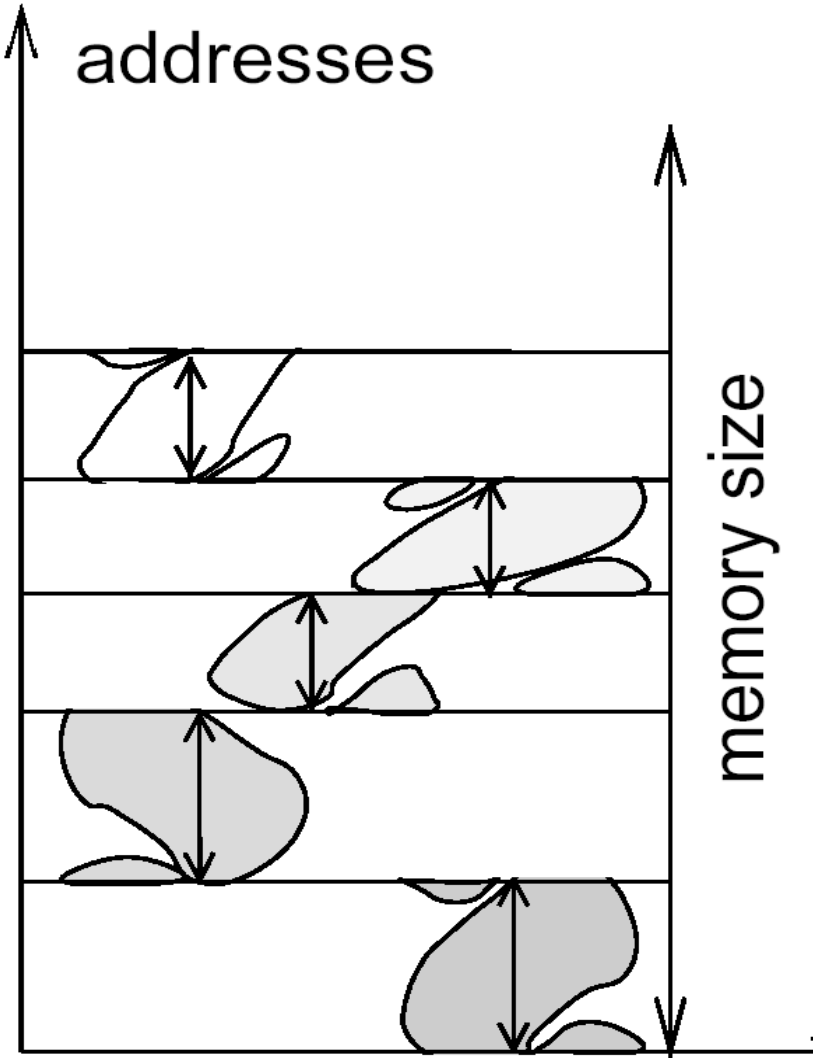
Unfolded arrays



Inter-array folding



Intra-array folding

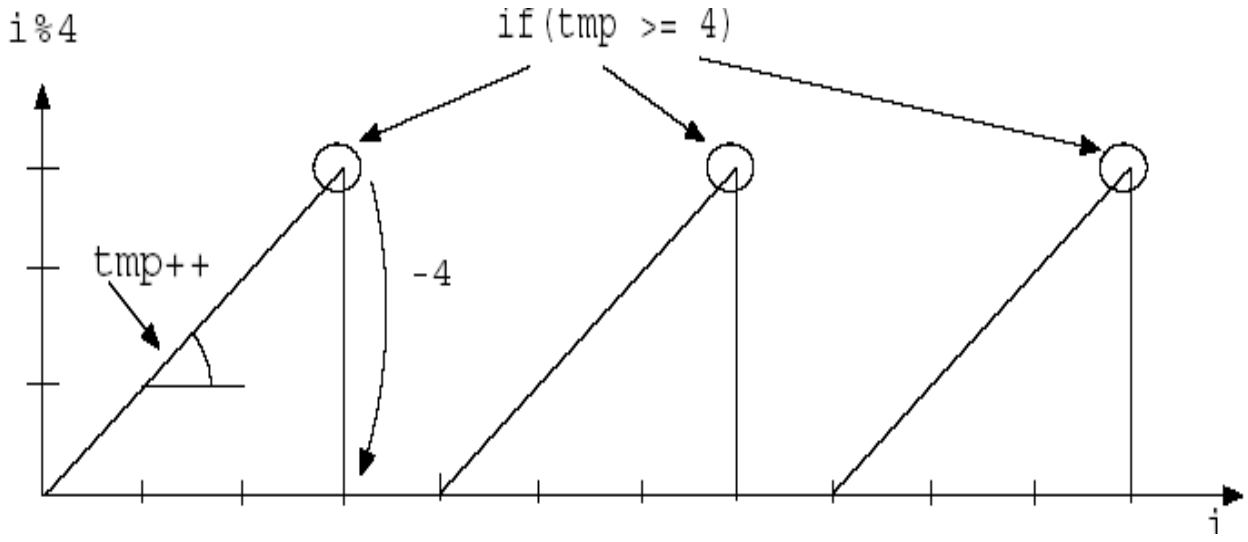


Application

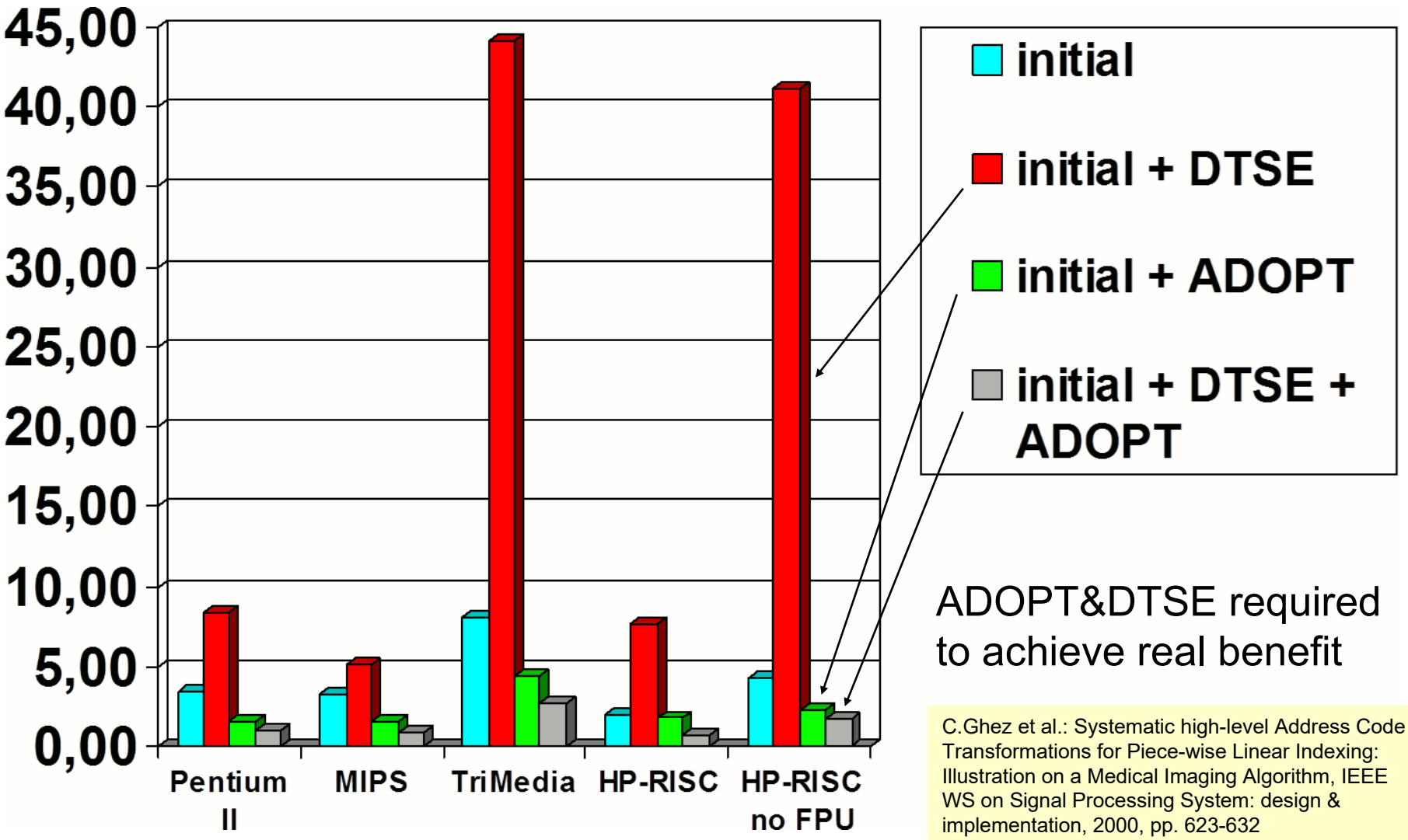
- Array folding is implemented in the DTSE optimization proposed by IMEC. Array folding adds div and mod ops. Optimizations required to remove these costly operations.
- At IMEC, ADOPT address optimizations perform this task. For example, modulo operations are replaced by pointers (indexes) which are incremented and reset.

```
for(i=0; i<20; i++)
      B[i % 4];

tmp=0;
for(i=0; i<20; i++)
      if(tmp >= 4)
         tmp -=4;
      B[tmp];
      tmp ++;
```



Results (Mcycles for cavity benchmark)



ADOPT&DTSE required to achieve real benefit

C.Ghez et al.: Systematic high-level Address Code Transformations for Piece-wise Linear Indexing: Illustration on a Medical Imaging Algorithm, IEEE WS on Signal Processing System: design & implementation, 2000, pp. 623-632

Summary

- Task concurrency management
  - Re-partitioning of computations into tasks
- Floating-point to fixed point conversion
  - Range estimation
  - Conversion
  - Analysis of the results
- High-level loop transformations
  - Fusion
  - Unrolling
  - Tiling
  - Loop nest splitting
  - Array folding

technische universität
dortmund

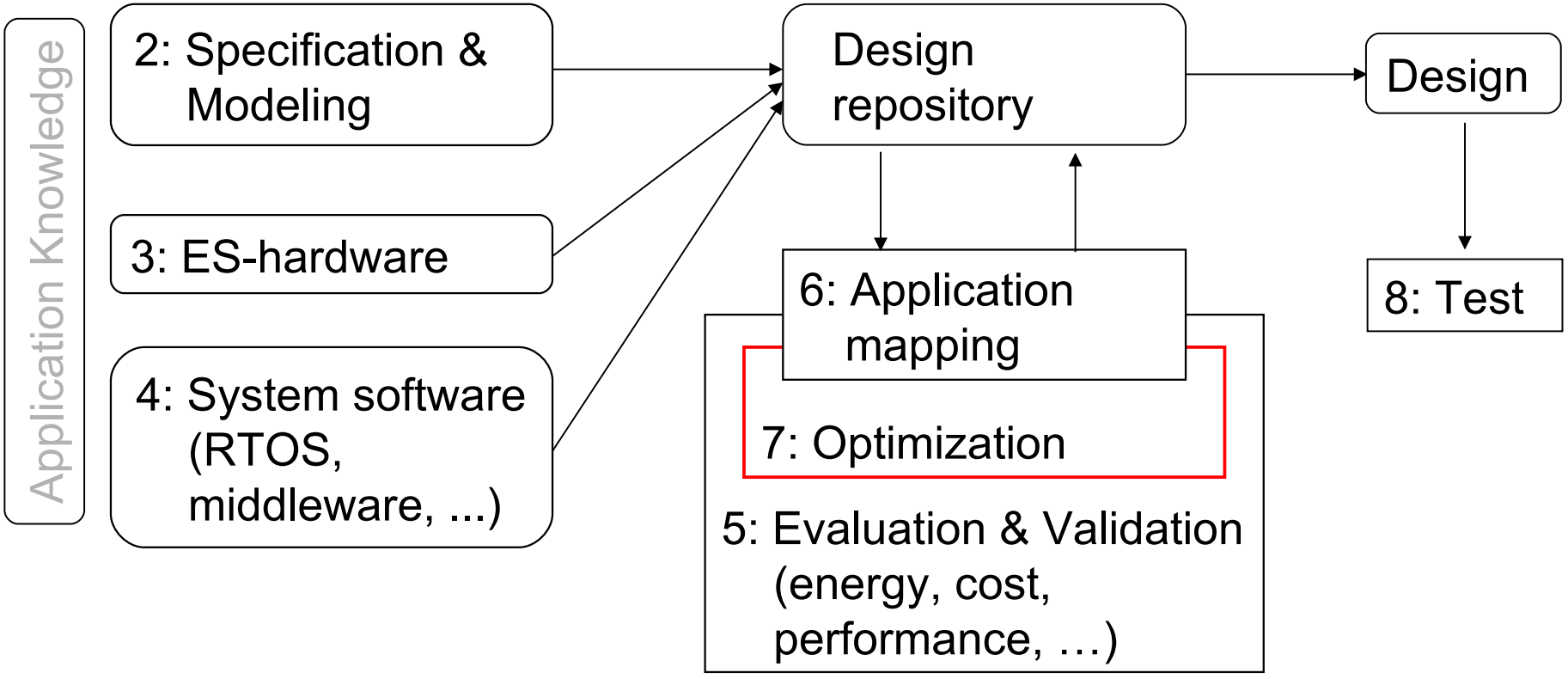
fakultät für informatik
informatik 12

Optimizations
- Compilation for Embedded Processors -

Peter Marwedel
TU Dortmund,
Informatik 12

2014 年 01 月 17 日

Structure of this course



Numbers denote sequence of chapters

Compilers for embedded systems: Why are compilers an issue?

- Many reports about low efficiency of standard compilers
  - Special features of embedded processors have to be exploited.
  - High levels of optimization more important than compilation speed.
  - Compilers can help to reduce the energy consumption.
  - Compilers could help to meet real-time constraints.
- Less legacy problems than for PCs.
  - There is a large variety of instruction sets.
  - Design space exploration for optimized processors makes sense

Compilers for embedded systems

Book section 7.3

- Introduction
- Energy-aware compilation
- Memory-architecture-aware compilation
- Reconciling compilers and timing analysis
- Compilation for digital signal processors
- Compilation for VLIW processors
- Compiler generation, retargetable compilers, design space exploration

Energy-aware compilation (1): Optimization for low-energy the same as for high performance?

No !

- High-performance if available memory bandwidth fully used; low-energy consumption if memories are at stand-by mode
- Reduced energy if more values are kept in registers

```
LDR r3, [r2, #0]
ADD r3,r0,r3
MOV r0,#28
LDR r0, [r2, r0]
ADD r0,r3,r0
ADD r2,r2,#4
ADD r1,r1,#1
CMP r1,#100
BLT LL3
```

2096 cycles
19.92 µJ

```
int a[1000];
c = a;
for (i = 1; i < 100; i++) {
 b += *c;
 b += *(c+7);
 c += 1;
}
```

2231 cycles
16.47 µJ

```
ADD r3,r0,r2
MOV r0,#28
MOV r2,r12
MOV r12,r11
MOV r11,rr10
MOV r0,r9
MOV r9,r8
MOV r8,r1
LDR r1, [r4, r0]
ADD r0,r3,r1
ADD r4,r4,#4
ADD r5,r5,#1
CMP r5,#100
BLT LL3
```

Energy-aware compilation (2)

- Operator strength reduction: e.g. replace * by + and <<
- Minimize the bitwidth of loads and stores
- Standard compiler optimizations with energy as a cost function

E.g.: Register pipelining

```
for i:= 0 to 10 do
    C:= 2 * a[i] + a[i-1];
```

```
R2:=a[0];
for i:= 1 to 10 do
  begin
    R1:= a[i];
    C:= 2 * R1 + R2;
    R2 := R1;
  end;
```

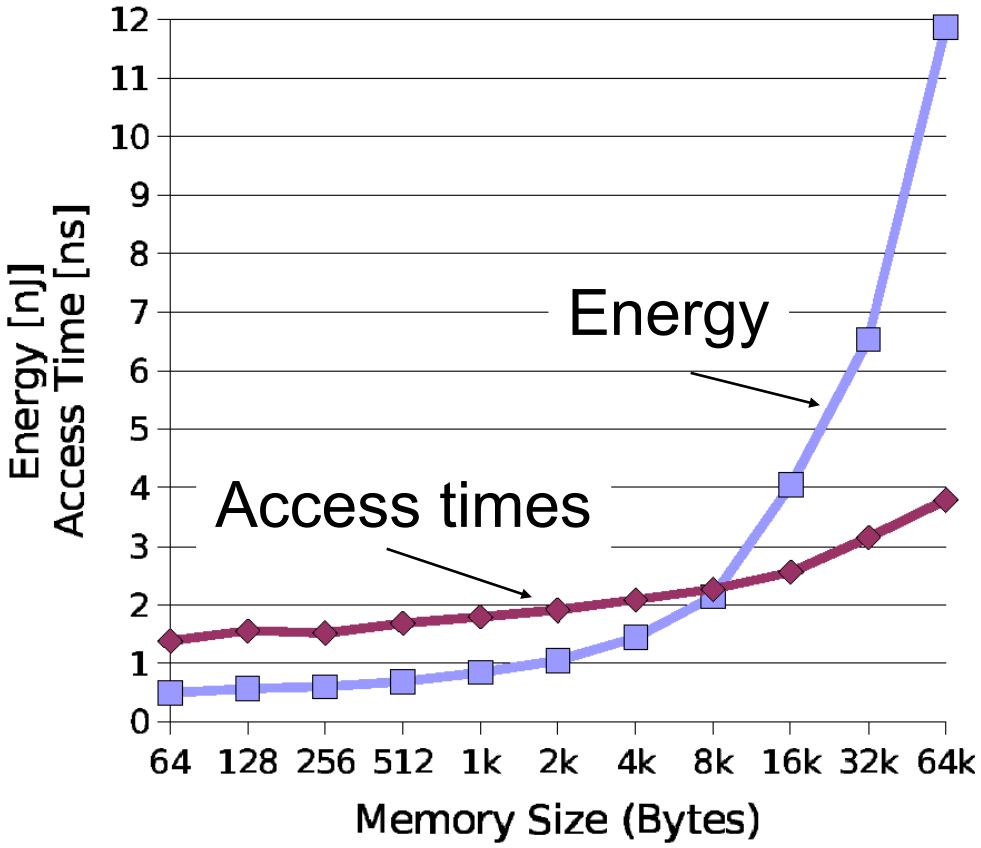
Exploitation of the memory hierarchy

Energy-aware compilation (3)

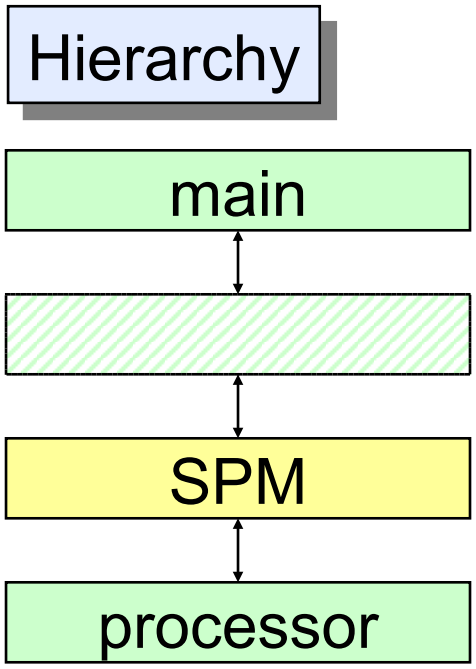
- Energy-aware **scheduling**: the order of the instructions can be changes as long as the meaning does not change. Goal: reduction of the number of signal transitions Popular (can be done as a post-pass optimization with no change to the compiler).
- Energy-aware **instruction selection**: among valid instruction sequences, select those minimizing energy consumption
- Exploitation of the **memory hierarchy**: huge difference between the energy consumption of small and large memories

3 key problems for future memory systems

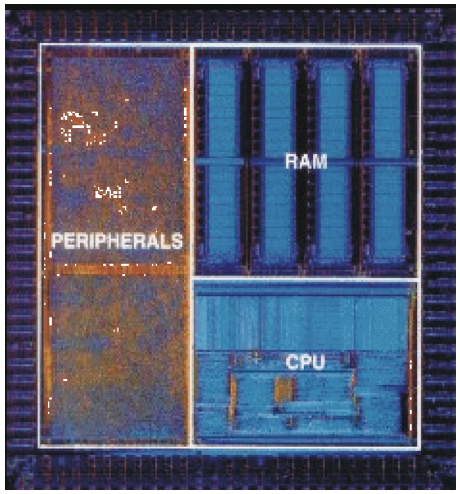
1. (average) Speed
2. Energy / Power
3. Predictability / WCET



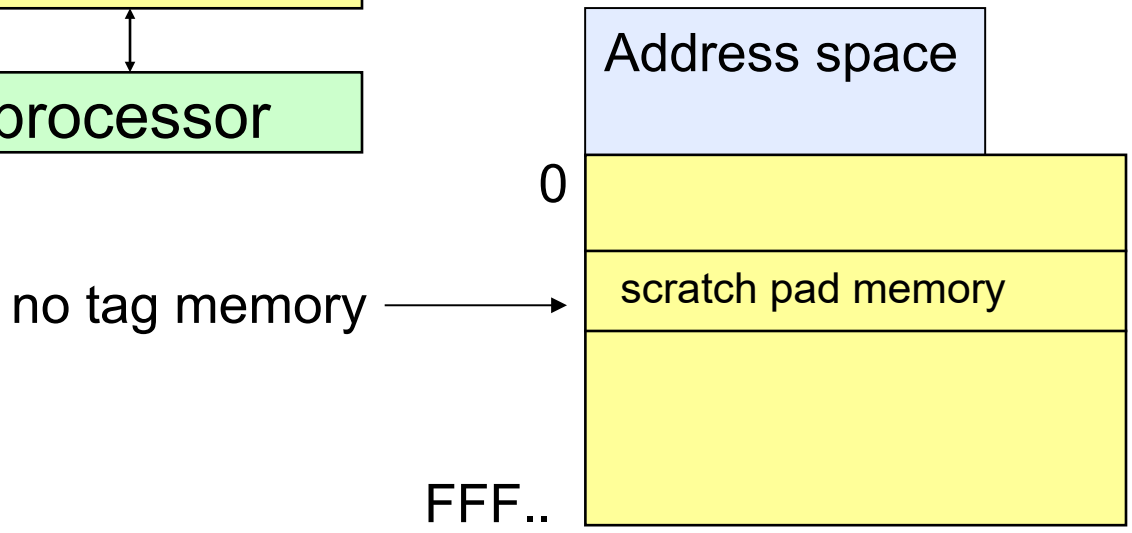
Hierarchical memories using scratch pad memories (SPM)



Example



ARM7TDMI cores, well-known for low power consumption



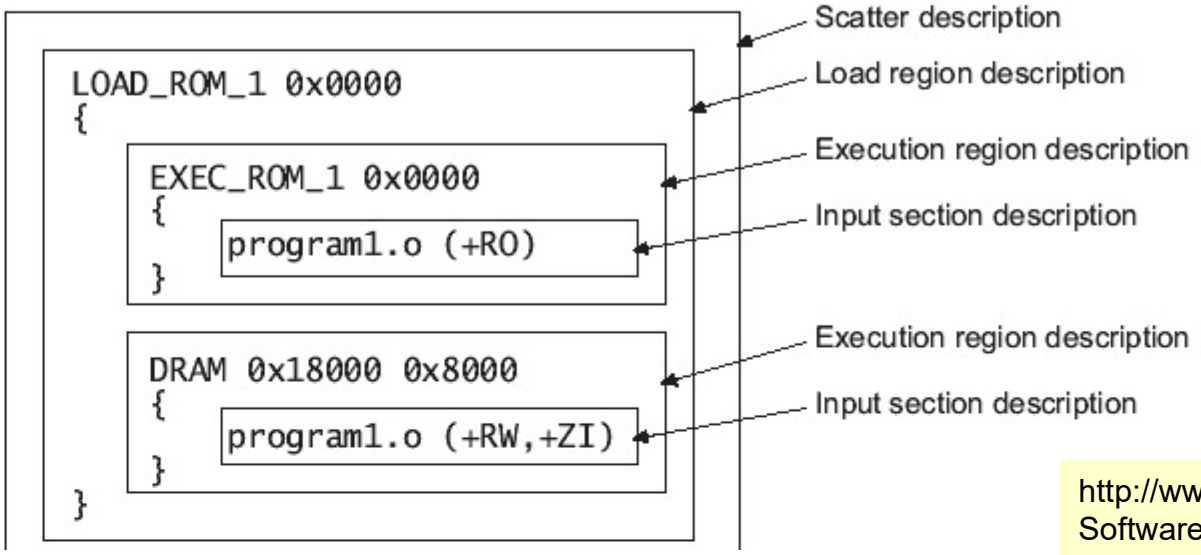
Very limited support in ARMcc-based tool flows

1. Use pragma in C-source to allocate to specific section

   For example:

```
#pragma arm section rwdata = "foo", rodata = "bar"
int x2 = 5;              // in foo (data part of region)
int const z2[3] = {1,2,3}; // in bar
```

2. Input scatter loading file to linker for allocating section to specific address range



http://www.arm.com/documentation/Software_Development_Tools/index.html

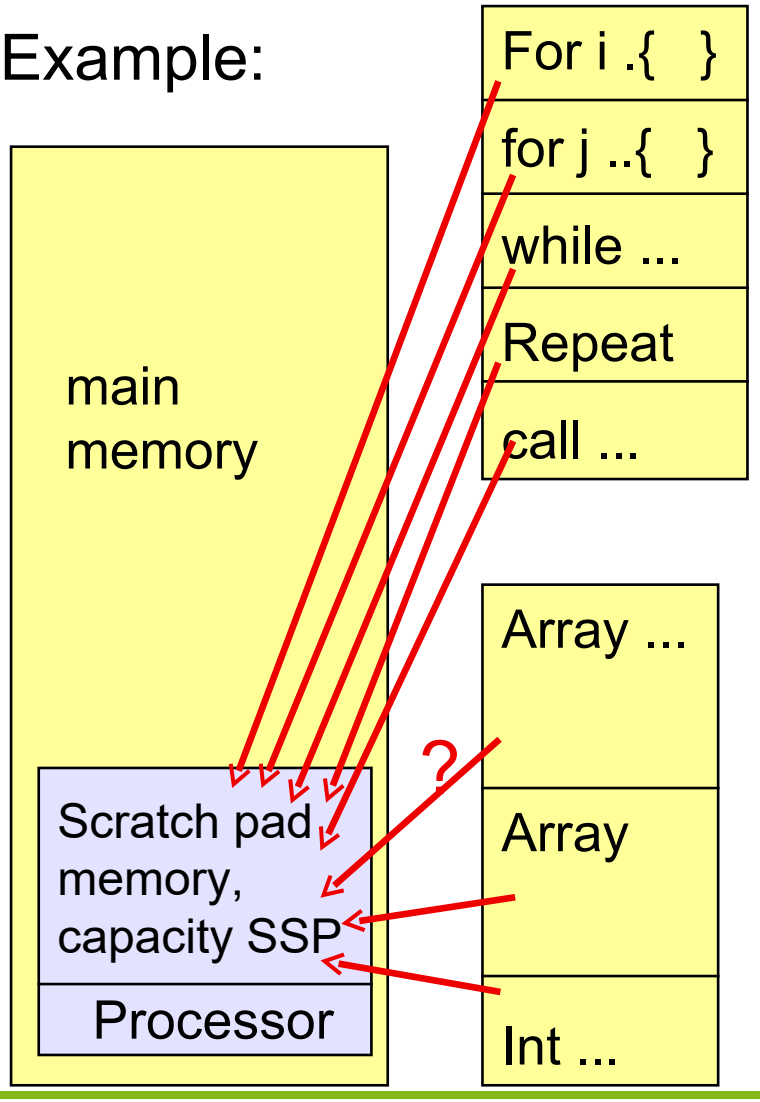
Compilers for embedded systems

Book section 7.3

- Introduction
- Energy-aware compilation
- Memory-architecture-aware compilation
- Reconciling compilers and timing analysis
- Compilation for digital signal processors
- Compilation for VLIW processors
- Compiler generation, retargetable compilers, design space exploration

Migration of data & instructions, global optimization model (TU Dortmund)

Example:



Which memory object (array, loop, etc.) to be stored in SPM?

1. **Non-overlaying ("static") allocation**
   - Gain g_k and size s_k for each object k.
   - Maximise gain $G = \Sigma g_k$, respecting size of SPM: $SSP \geq \Sigma\, s_k$.
   - Solution: knapsack algorithm.
2. **Overlaying ("dynamic") allocation**
   - Moving objects back and forth

ILP representation
- migrating functions and variables-

Symbols:

$S(var_k)$ = size of variable k

$n(var_k)$ = number of accesses to variable k

$e(var_k)$ = energy **saved** per variable access, if var_k is migrated

$E(var_k)$ = energy **saved** if variable var_k is migrated ($= e(var_k)\ n(var_k)$)

$x(var_k)$ = decision variable, =1 if variable k is migrated to SPM,
=0 otherwise

K = set of variables; similar for functions I

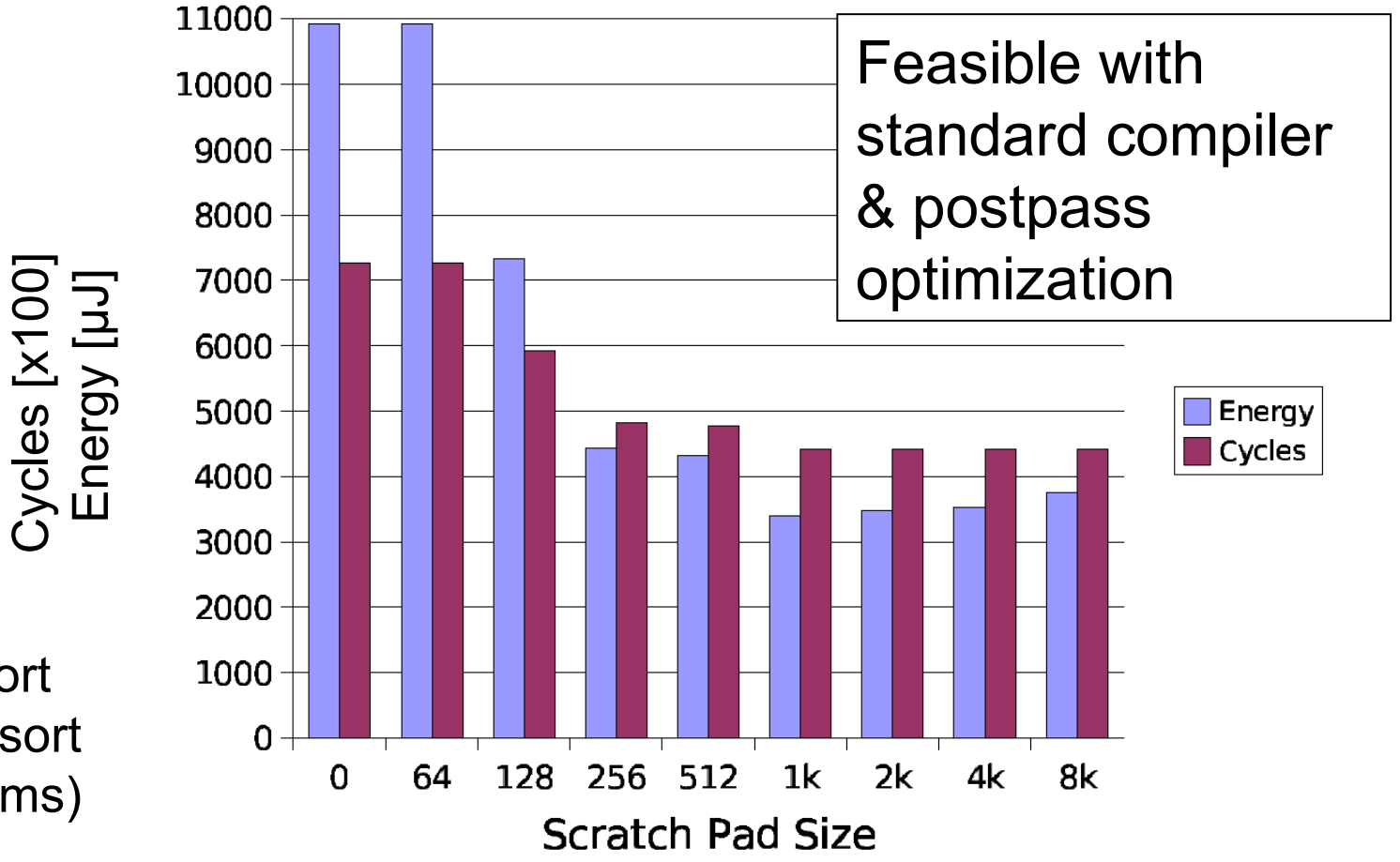
Integer programming formulation:

Maximize $\sum_{k \in K} x(var_k)\ E(var_k) + \sum_{i \in I} x(F_i)\ E(F_i)$

Subject to the constraint

$\sum_{k \in K} S(var_k)\ x(var_k) + \sum_{i \in I} S(F_i)\ x(F_i) \leq SSP$

Reduction in energy and average run-time



Feasible with standard compiler & postpass optimization

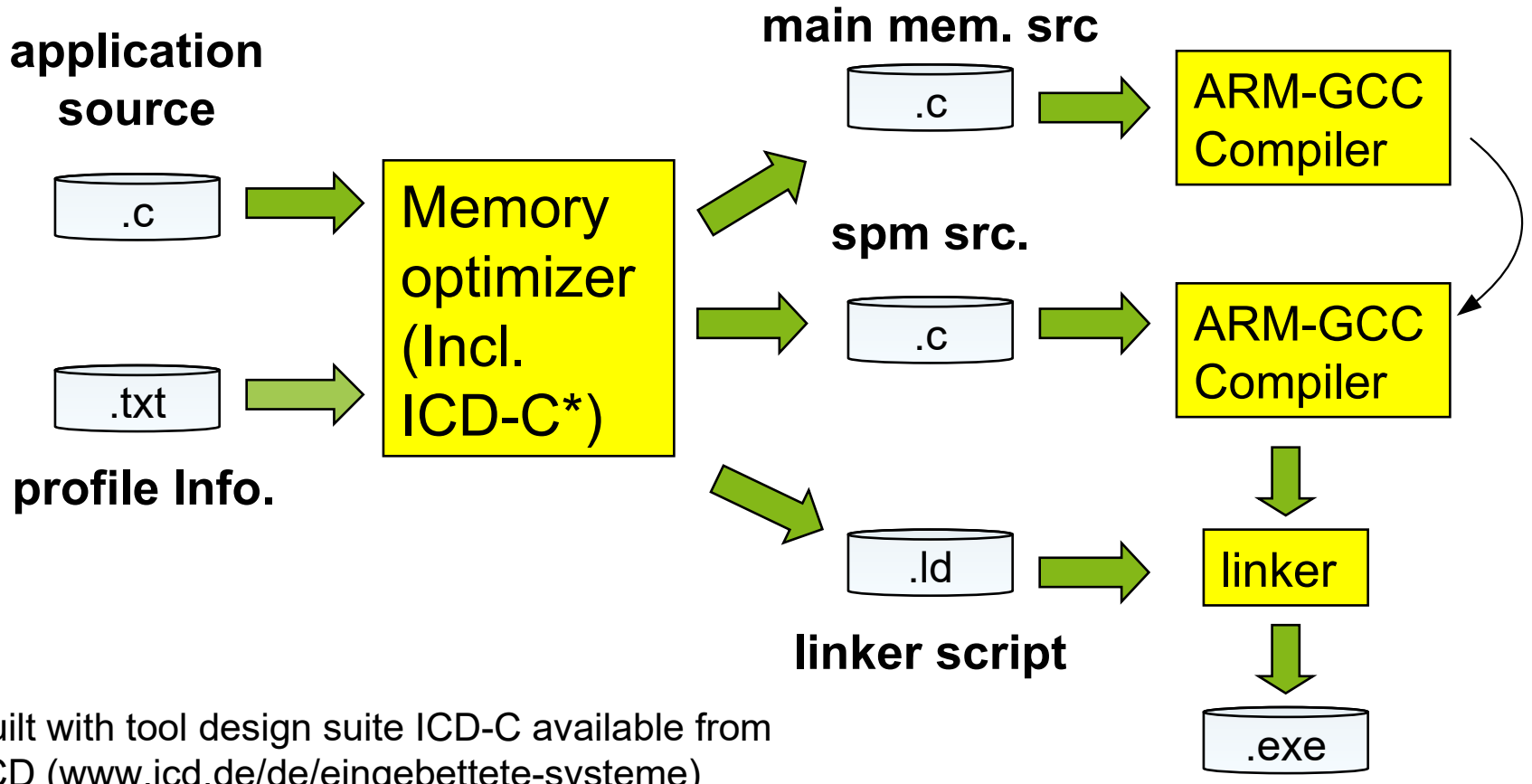
Multi_sort (mix of sort algorithms)

Measured processor / external memory energy + CACTI values for SPM (combined model)

Numbers will change with technology, algorithms remain unchanged.

Using these ideas with an gcc-based tool flow

Source is split into 2 different files by specially developed memory optimizer tool *

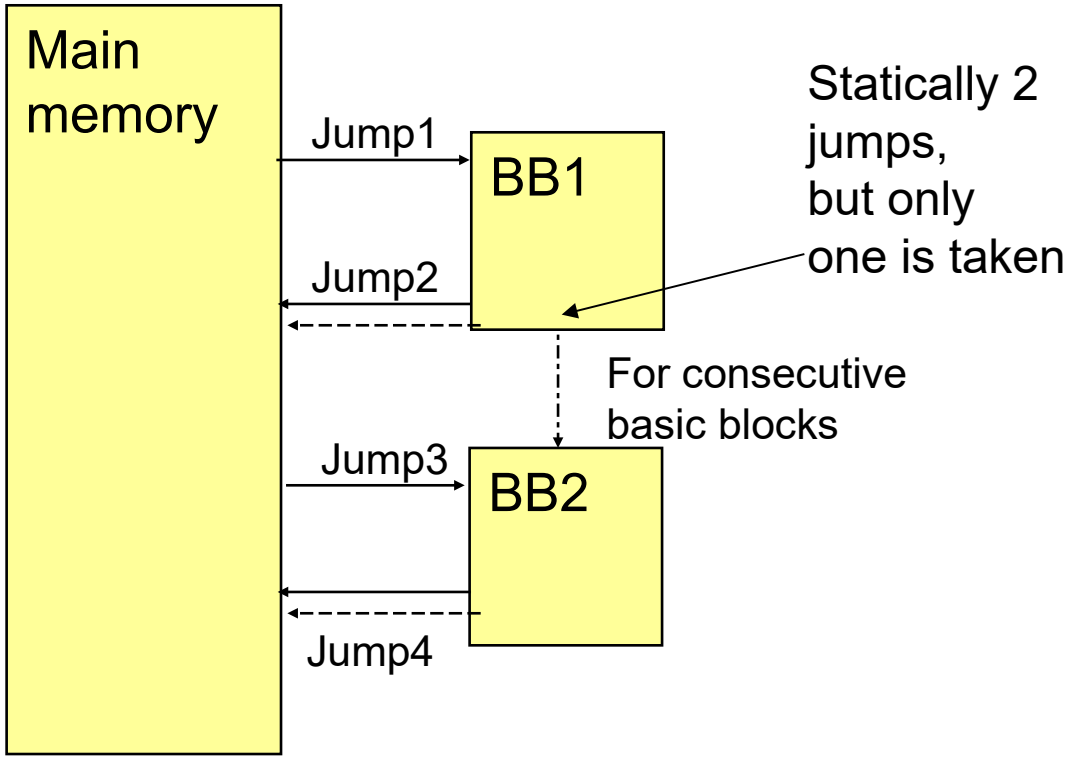


* Built with tool design suite ICD-C available from ICD (www.icd.de/de/eingebettete-systeme)

Allocation of basic blocks

Fine-grained granularity smoothens dependency on the size of the scratch pad.

Requires additional jump instructions to return to "main" memory.



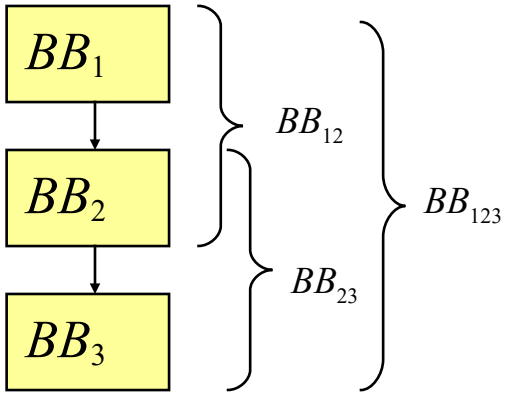
Taking consecutive basic blocks into account

Approach:

- Consider sets of consecutive BBs as a new kind of basic blocks ("multi blocks")
- Add a constraint preventing the same block from being selected twice:

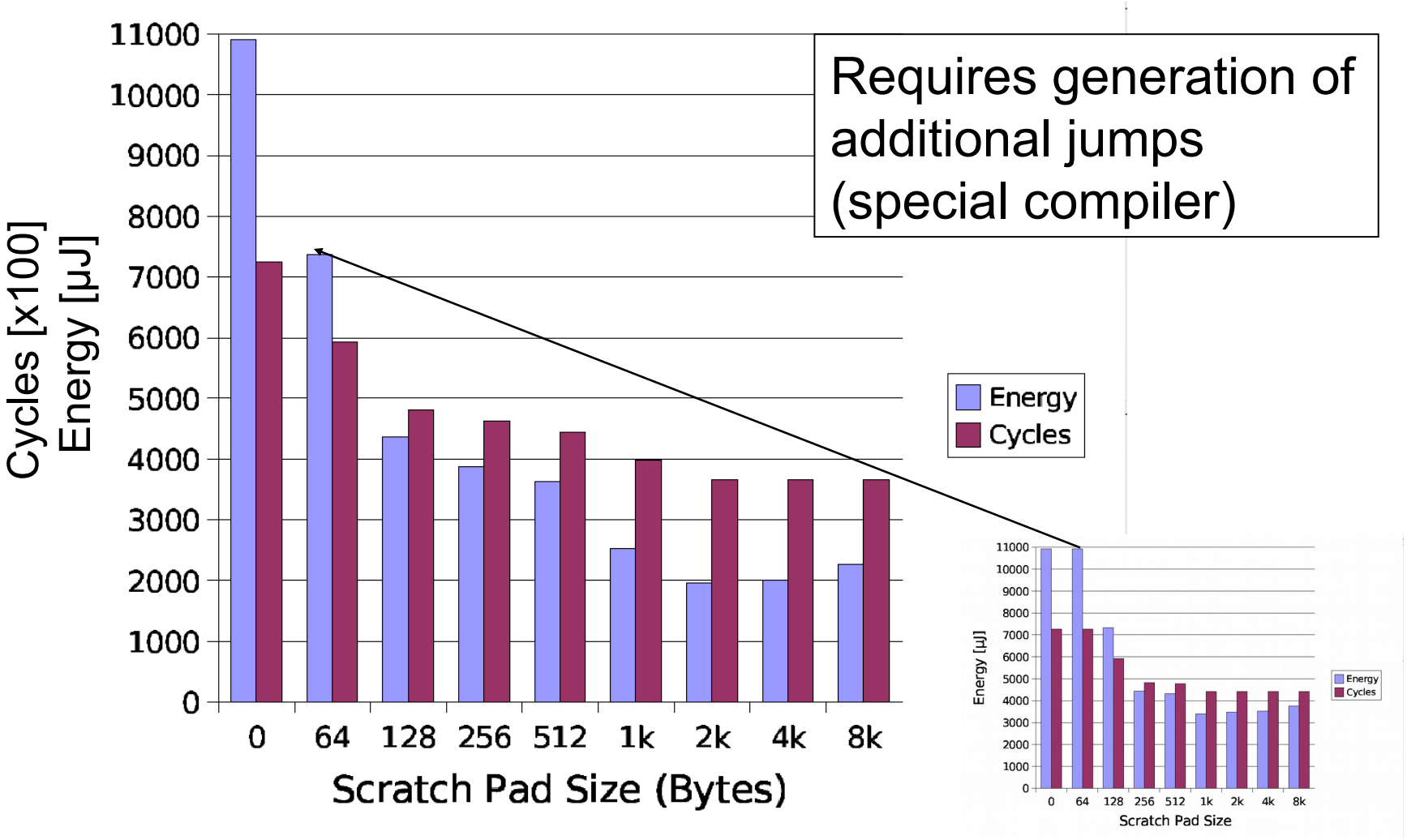
$$x(BB_b) + x(F_i) + \sum_{j \in multiblocks(b)} x(BB_j) \leq 1$$

$$\forall\, b \in \{blocks\} \cup \{\text{multi blocks}\}$$

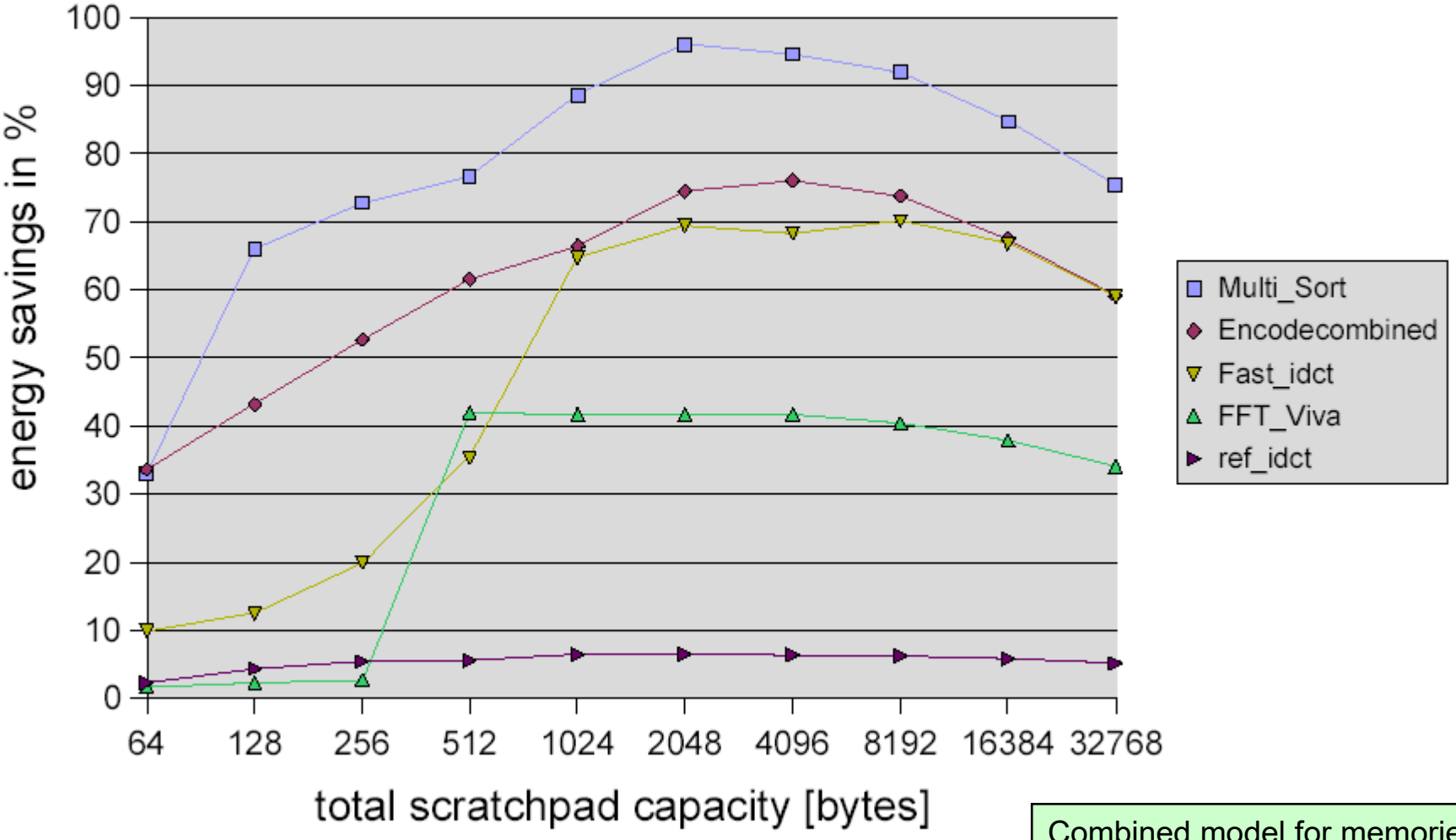


☞ Block b is either moved individually, as part of a function, as part of one of its enclosing multi-blocks or not at all.

Allocation of basic blocks, sets of adjacent basic blocks and the stack



Savings for memory system energy alone



Combined model for memories

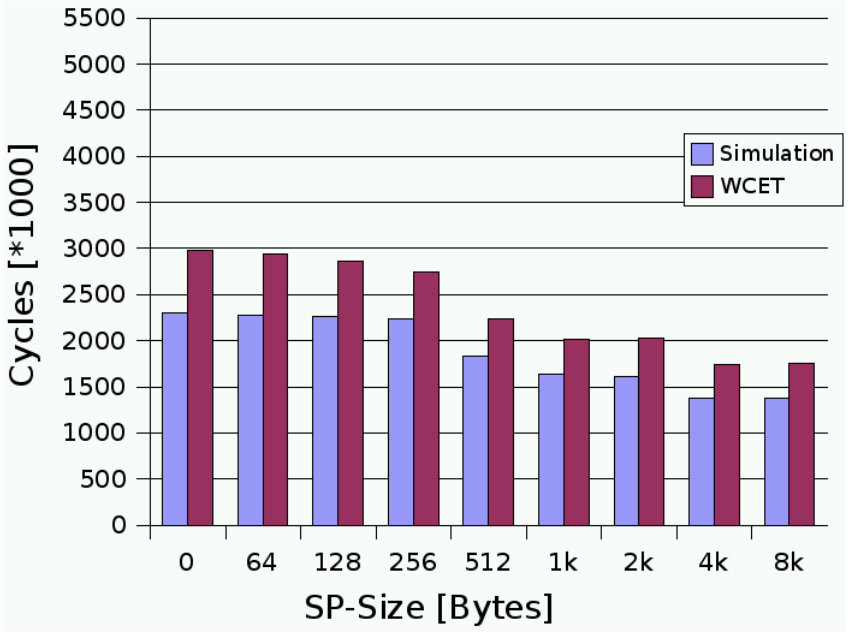
Architectures considered

ARM7TDMI with 3 different memory architectures:

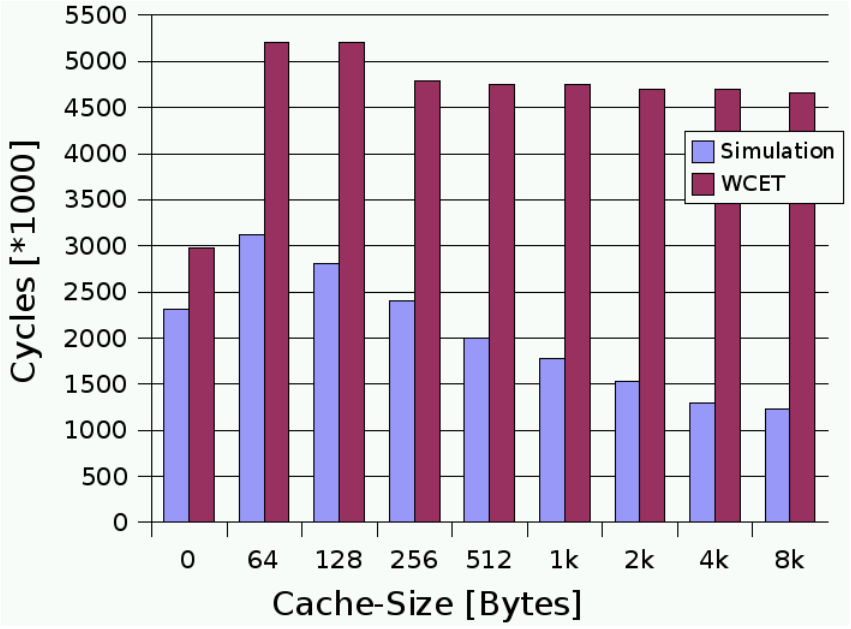
1. **Main memory**
   LDR-cycles: (CPU,IF,DF)=(3,2,2)
   STR-cycles: (2,2,2)
   * = (1,2,0)
2. **Main memory + unified cache**
   LDR-cycles: (CPU,IF,DF)=(3,12,6)
   STR-cycles: (2,12,3)
   * = (1,12,0)
3. **Main memory + scratch pad**
   LDR-cycles: (CPU,IF,DF)=(3,0,2)
   STR-cycles: (2,0,0)
   * = (1,0,0)

Results for G.721

Using Scratchpad:



Using Unified Cache:



References:

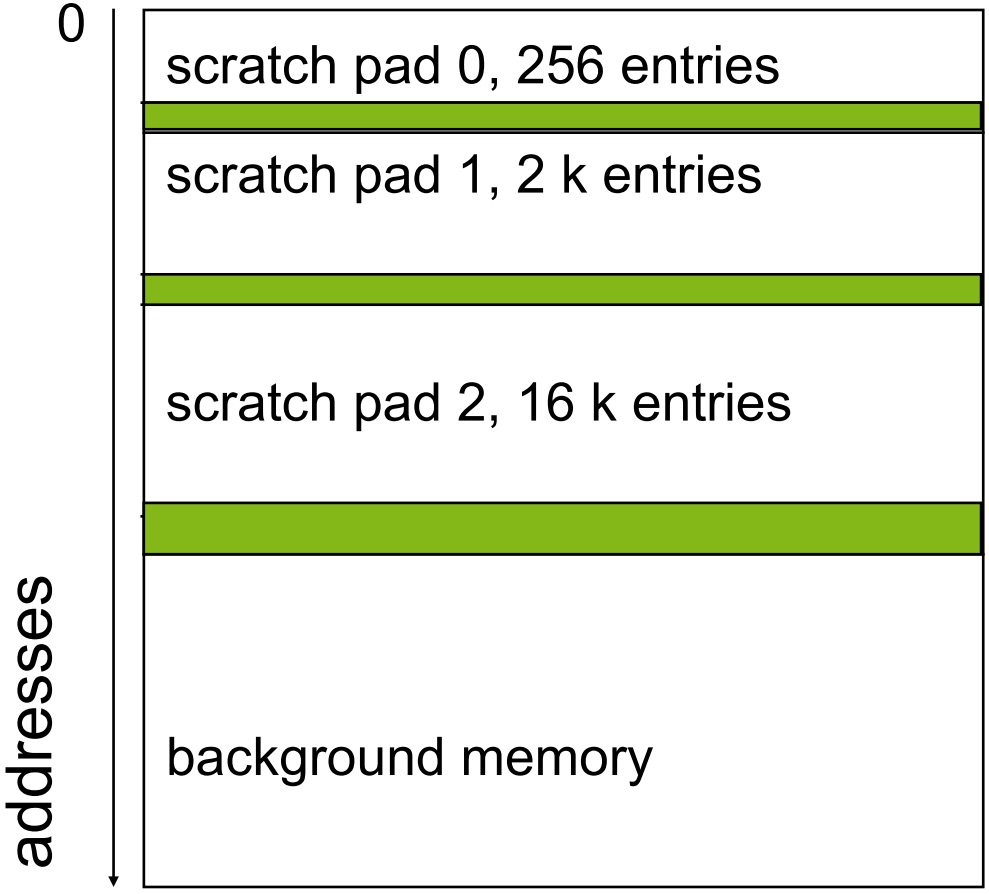
- Wehmeyer, Marwedel: Influence of Onchip Scratchpad Memories on WCET: 4th Intl Workshop on worst-case execution time (WCET) analysis, Catania, Sicily, Italy, June 29, 2004
- Second paper on SP/Cache and WCET at DATE, March 2005

Multiple scratch pads

Small is beautiful:

One small SPM is beautiful (☺).

May be, several smaller SPMs are even more beautiful?



Optimization for multiple scratch pads

Minimize $$C = \sum_j e_j \cdot \sum_i x_{j,i} \cdot n_i$$

With e_j: energy per access to memory j,

$x_{j,i}$= 1 if object i is mapped to memory j, =0 otherwise,

n_i: number of accesses to memory object i,

subject to the constraints:

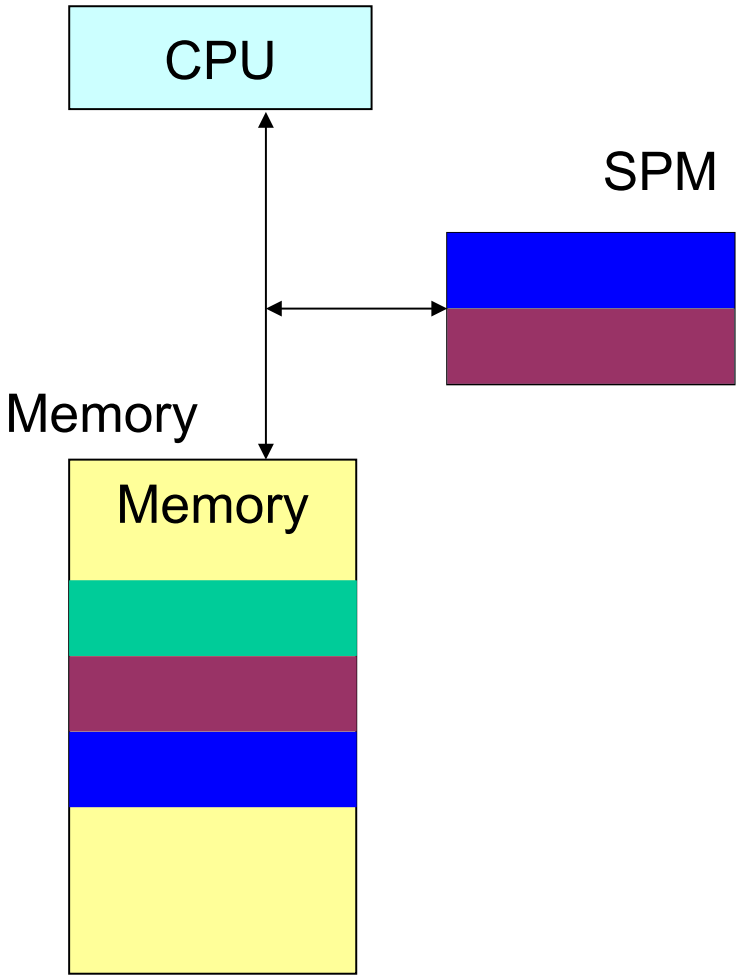
$$\forall j : \sum_i x_{j,i} \cdot S_i \leq SSP_j$$

$$\forall i : \sum_j x_{j,i} = 1$$

With S_i: size of memory object i,

SSP_j: size of memory j.

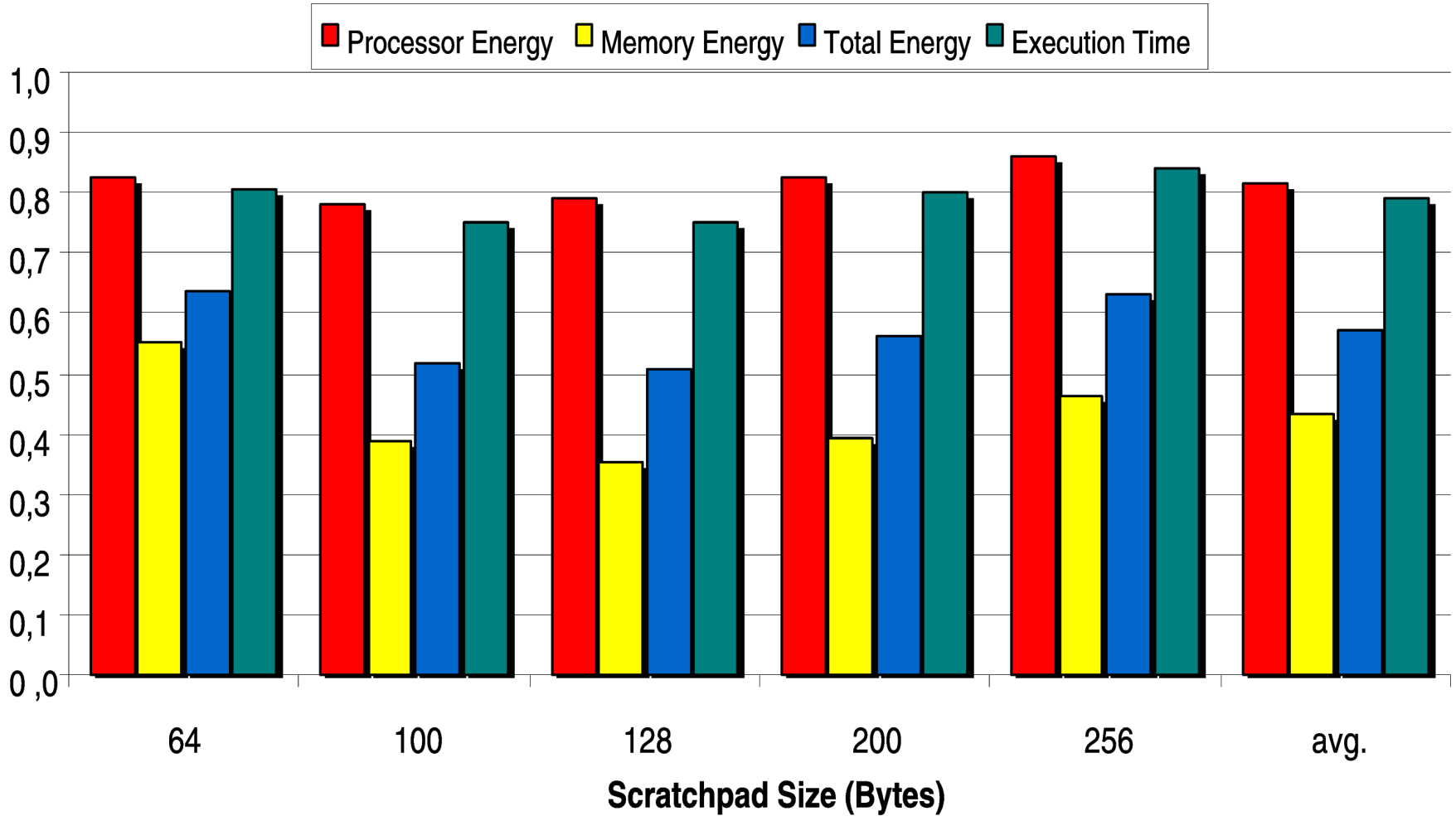
Dynamic replacement within scratch pad



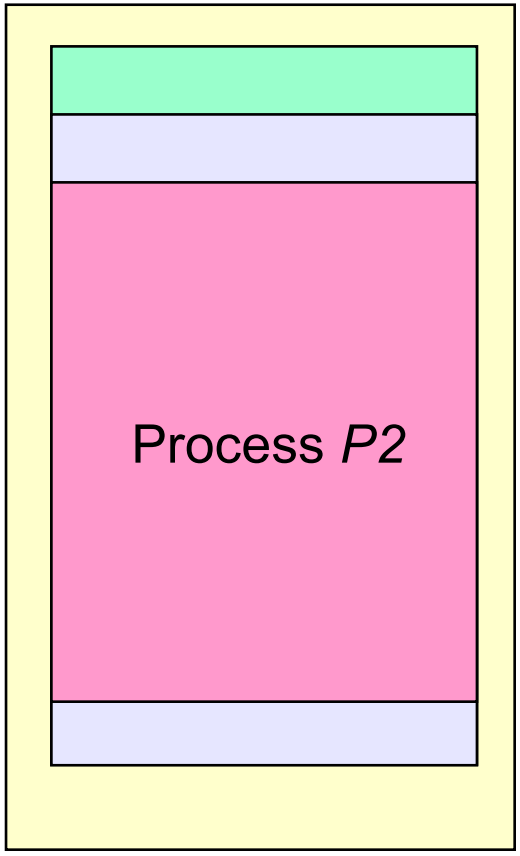
- Effectively results in a kind of **compiler-controlled segmentation / paging** for SPM
- Address assignment within SPM required (paging or segmentation-like)

Reference: Verma, Marwedel: Dynamic Overlay of Scratchpad Memory for Energy Minimization, ISSS 2004

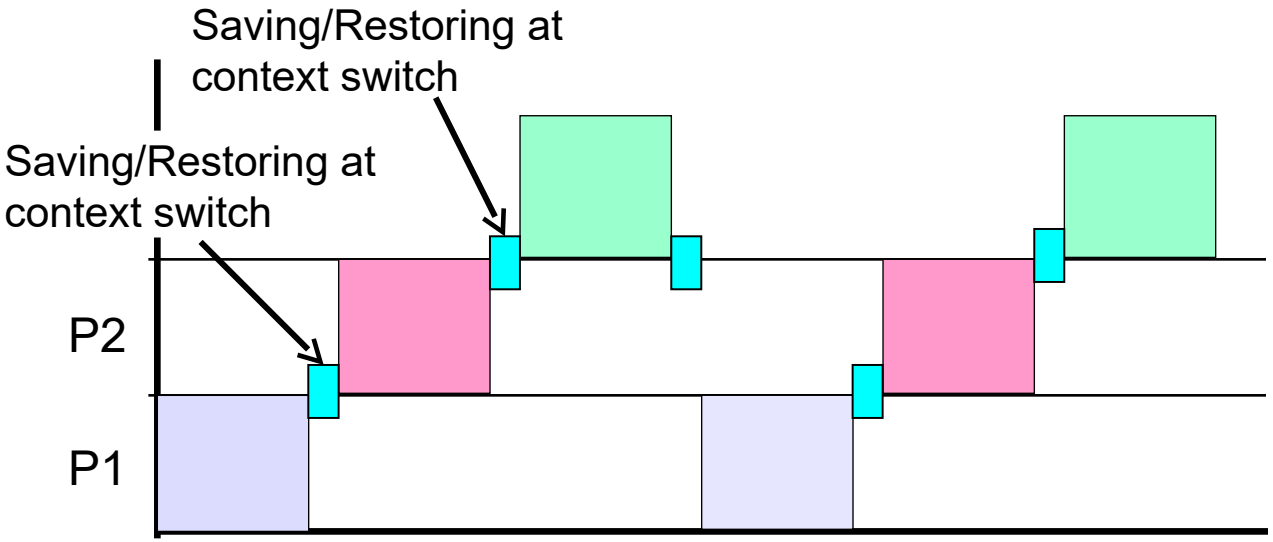
Dynamic replacement within SPM
Edge detection relative to non-overlaying/static allocation



Saving/Restoring Context Switch



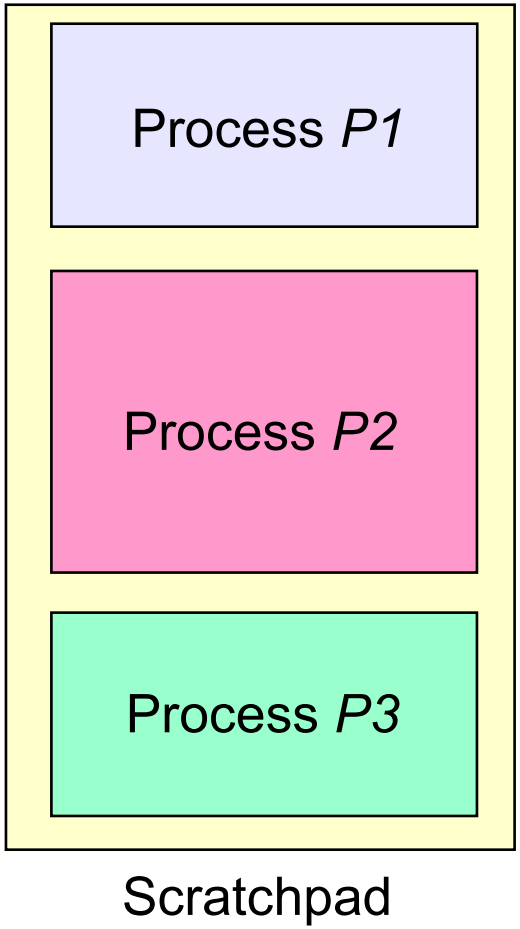
Scratchpad

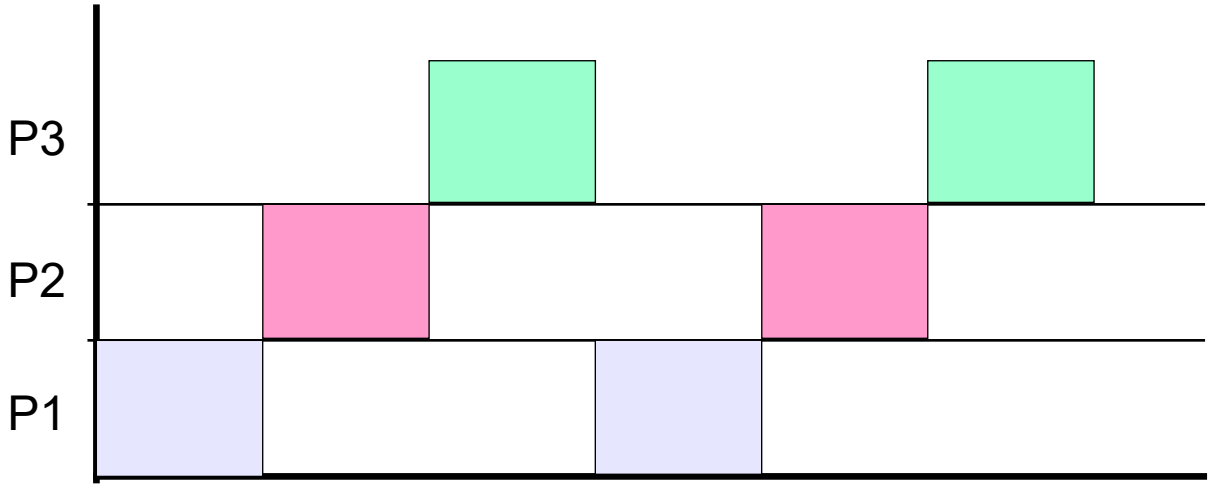


Saving Context Switch (Saving)

- Utilizes SPM as a common region shared all processes
- Contents of processes are copied on/ off the SPM at context switch
- Good for small scratchpads

Non-Saving Context Switch

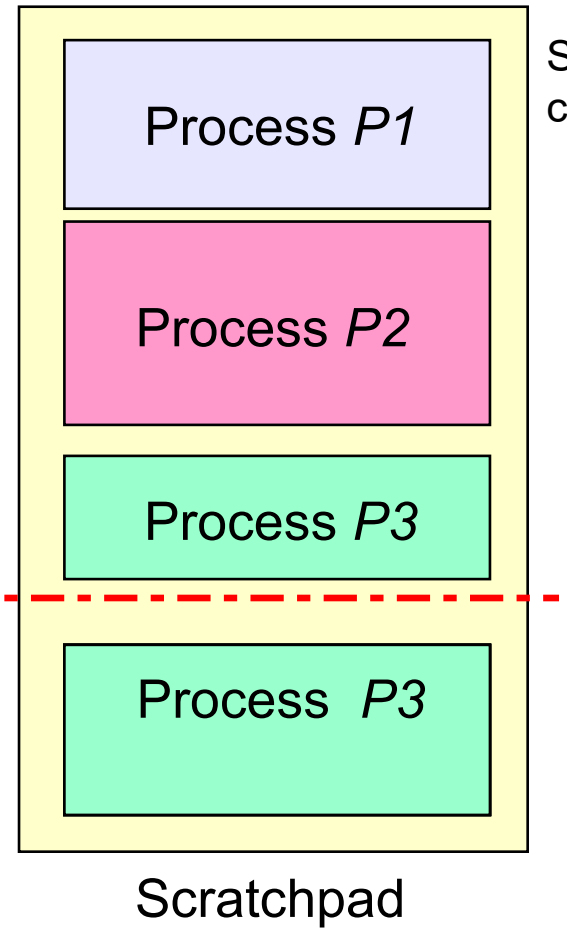


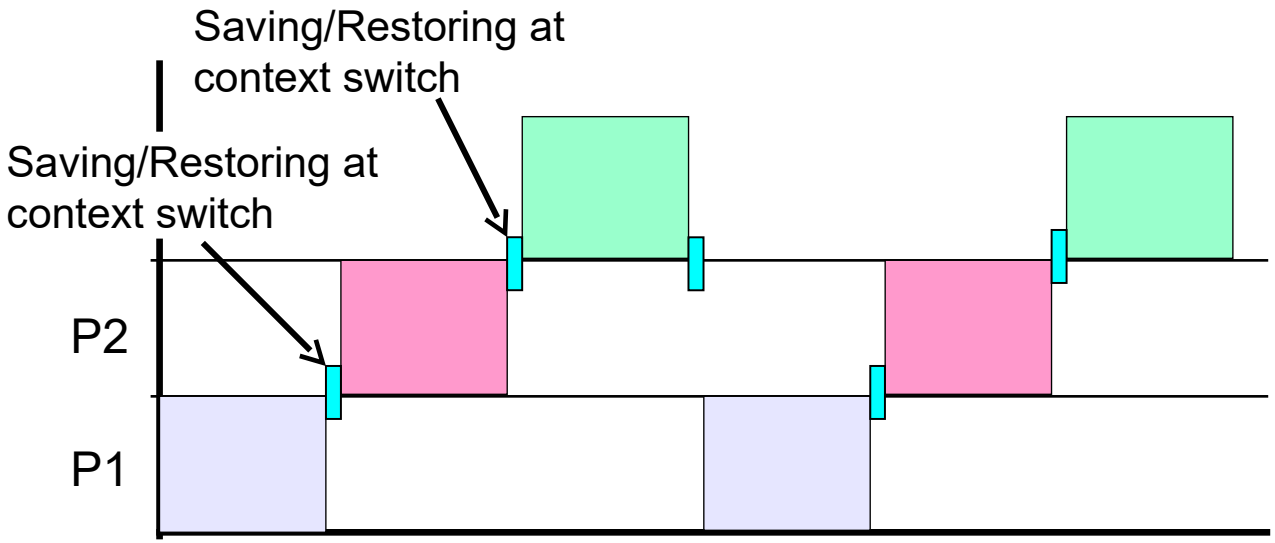


Non-Saving Context Switch

- Partitions SPM into disjoint regions
- Each process is assigned a SPM region
- Copies contents during initialization
- Good for large scratchpads

Hybrid Context Switch

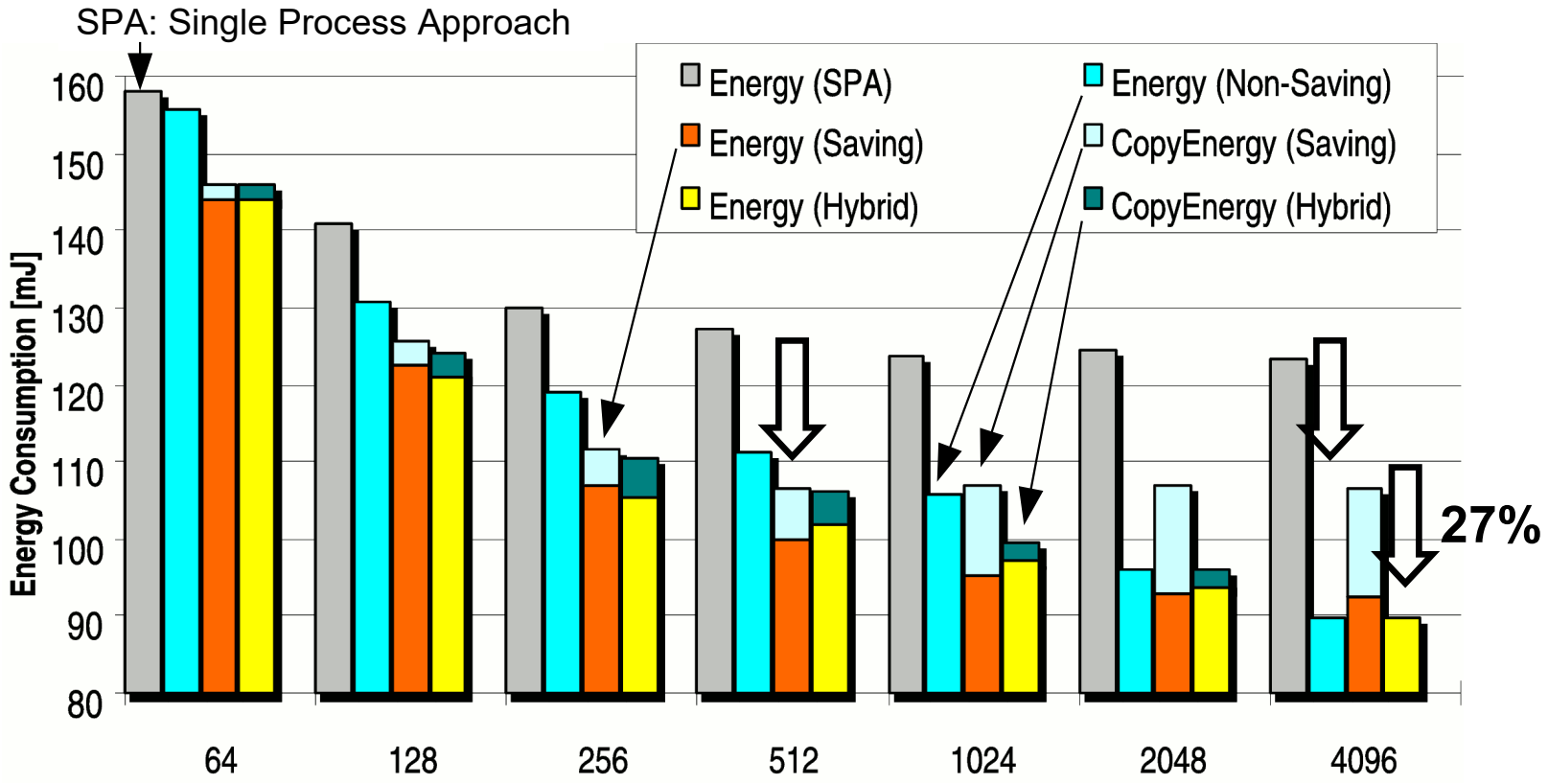




Hybrid Context Switch (Hybrid)

- Disjoint + Shared SPM regions
- Good for all scratchpads
- Analysis is similar to Non-Saving Approach
- Runtime: $O(nM^3)$

Multi-process Scratchpad Allocation: Results



- For small SPMs (64B-512B) Saving is better
- For large SPMs (1kB- 4kB) Non-Saving is better
- Hybrid is the best for all SPM sizes.
- Energy reduction @ 4kB SPM is 27% for Hybrid approach

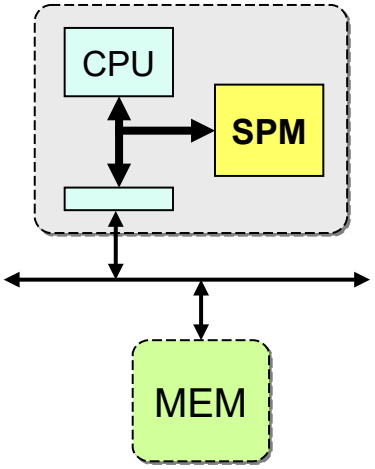
edge detection, adpcm, g721, mpeg

Dynamic set of multiple applications

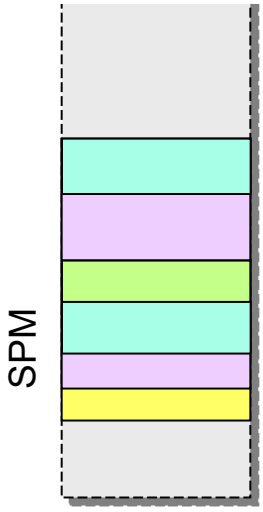
Compile-time partitioning of SPM no longer feasible

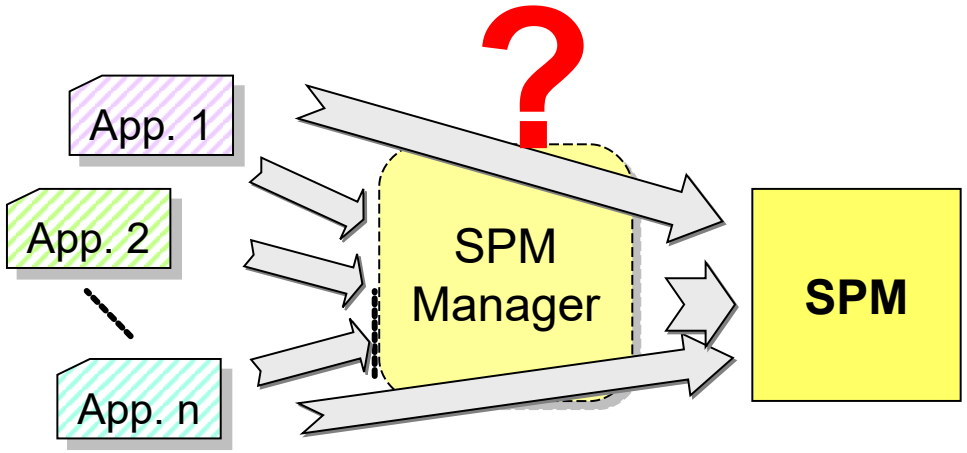
☞Introduction of SPM-manager

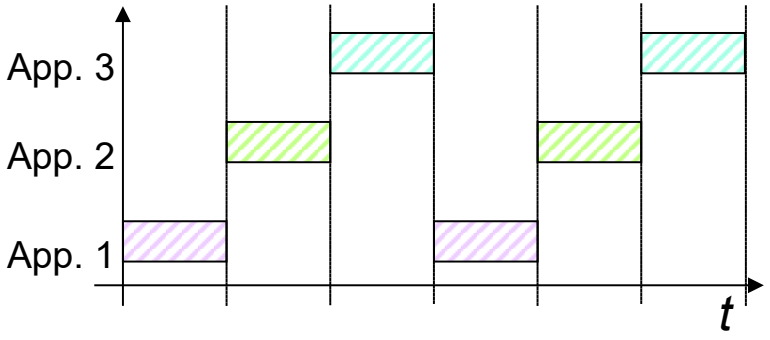
- Runtime decisions, but compile-time supported



Address space:







R. Pyka, Ch. Faßbach, M. Verma, H. Falk, P. Marwedel: Operating system integrated energy aware scratchpad allocation strategies for multi-process applications, *SCOPES*, 2007

Summary

Impact of memory architecture on execution times and energy consumption

- The SPM provides
  - Runtime efficiency, energy efficiency, timing predictability
- Allocation strategies
  - Static allocation
    - Partitioning
    - Timing predictability
  - Dynamic allocation
    - Tiling
    - Multiple hierarchy levels
    - Multiple processes
    - Dynamic sets of processes
- Savings dramatic, e.g. ~ 95% of the memory energy