# Predictability Issues in Operating Systems

Space, Timing, Energy

*Wolfgang Schröder-Preikschat*
*Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)*



Predictability refers to the degree to which a correct quantitative or qualitative prediction of the state of a system can be made. In the following considerations, this state relates to operating systems. Central role plays time behavior, which is not only determined by the external processes to a given frame of reference but also influenced by spatial and energetic characteristics of the system software therein.






In animation order:

1. This lecture has two purposes. On the one hand it wants to clarify some technical challenges of a balancing act in the design and development of operating systems. On the other hand it would like to build a bridge to the outside and thereby promote understanding of certain non-functional features in such complexes.
2. Features that influence the time behavior of a system, delay processes unintentionally, cause uncertainties and thus let time to act melt away.
3. These features have cause in system functions that are gathered together in the same frame of reference and which depend directly or indirectly on common resources.

# Bifocal perspective

issue (acc. Webster, i.a.):

- *a point, matter, or question to be disputed or decided*
- problems on the one hand
  - resource usage/use
  - interference
- aspects on the other hand
  - software structuring
  - implementation




In animation order:

1. The issues I want to pursue refer to two points, matters, or questions in the design and development of non-sequential system programs in general and operating systems in specific.
2. On the one hand they raise certain problems due to a shared use of resources. Especially problems that cause interference.
3. On the other hand these problems must be taken as a fact due to software-structuring measures and implementation decisions. They have to be accepted as given non-functional properties of a particular system — and therefore should be externalised and made part of the application binary interface (ABI) of an operating system.

# Space

- memory demand
  - static
  - dynamic
- stack usage
  - best/worst case
- use pattern
  - process locality
  - data-structure alignment

Speicherstadt („city of warehouses") in Hamburg

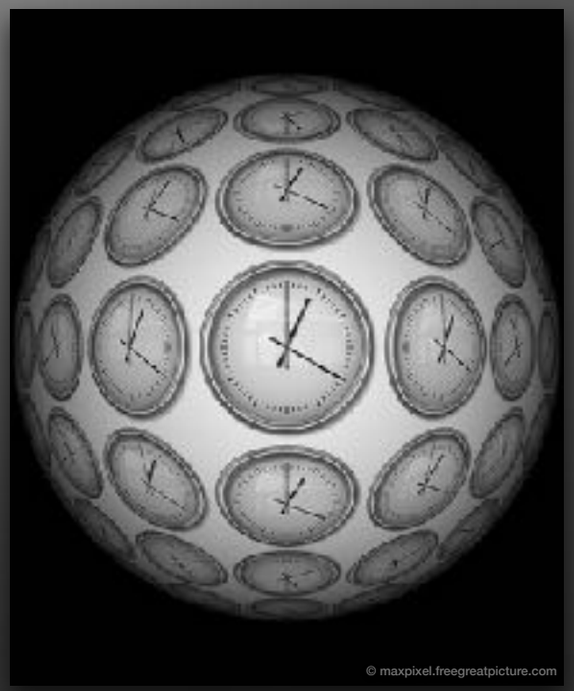



The first point dealt with relates to space.

In animation order:
1. The space aspect concerns the memory demand or storage requirements of a process. This issue is of static (simple) or dynamic (complex) nature.
2. A special point is here stack usage (dynamic), as its correct quantitative prediction has an influence on the reliability not only of an individual process but also of the whole computing system. The best case (smallest need) saves resources, while the worst case (biggest need) sets the safe side. Both cases ultimately save money — for hardware facilities on the one hand and insurance protection on the other hand.
3. For the purpose of the prediction, knowledge about the usage pattern is very appropriate. Particularly knowledge about process locality and data-structure alignment, two important points to make estimations about the time behavior of a certain set of entities unknowingly interacting with each other — keyword cache.

# Timing

- timeliness
- target deadlines
  - soft: violation is tolerated, task continues
  - firm: violation is tolerated, task is aborted
  - hard: violation is not tolerated, exception is raised, *safe state* is to be taken
- latency






So one can deduce that the space aspect may influence timing and, thus, the ability for timeliness.

In animation order:

1. However to be ready in time does not mean being fast, but it means a certain amount of assurance for a set of processes (tasks) that each of them meets a specified deadline.
2. Each of those target deadlines is qualified by a grade, whereby either a two-stage (soft, hard) or a three-stage (soft, firm, hard) distinction is made. This grading from soft over firm to hard does not necessarily correspond to the level of difficulty in the implementation of a certain real-time property. The longer a deadline has passed, the less weight the result calculated too late has. From that follows that soft deadlines need to be monitored even after they were missed and corresponding weightings are to be updated. Such a system function is not required for firm or hard deadlines — whereas cancelling of tasks and raising of exceptions can be already remarkably easy actions of an operating system.
3. Finally, the period between an event and the subsequent reaction in real time. This period must be bounded and must be subject to little or no jitter.

# Energy

- power demand
  - ecological aspect
  - technical constraint
  - economical factor
- thermal dissipation
  - dark silicon
- power-band observance
  - contractual obligations




Closely related to time is energy, that is to say, the energy converted over a period of time in relation to this period of time — performance.

In animation order:

1. The power demand ultimately produced by a process is not only a technical constraint, but from a certain amount also of economical and not least ecological importance
2. A further specific aspect thereby is thermal dissipation, which leads to dark silicon in large-scale many-core processors (MPSoC).
3. At the other end of the spectrum is the need for power-band observance in data or computing centres to avoid costly violations of contractual obligations with energy suppliers. The power demand should never exceed a certain upper limit, but it should also not fall below a certain lower limit. Thus, while power saving is always good for ecological and reasonable for specific technical reasons, it is occasionally not first choice for economical reasons.

# Preliminary remark

- This talk is *not* about analytical methods to predetermine quality attributes
  - of non-sequential (real-time) processes
- but about structuring principles
  - of non-sequential programs

to favor predetermination of these attributes.

# Space

Memory footprint

**“Some users may require only a subset of the services or features that other users need. These ‘less demanding’ users may demand that they not be forced to pay for the resources consumed by the unneeded features.”**

***–David Parnas, 1979****

***Designing Software for Ease of Extension and Contraction, IEEE TSE, vol. SE-5, no. 2**



The memory footprint of an operating system stands and falls with the function to be provided for the respective application or class of applications. There is no one size fits all solution.

# Customisation

- modularisation for the purpose of a program family
  - a “software product line” in modern terms

*cross-cutting concerns*

- reuse and adaptation need to go hand in hand
  - above all, apply compositional and generic approaches
    - “aspect-oriented programming” in the broader sense
  - decompositional methods (`#ifdef`) have to be handled with care

- establish the basis for the non-functional properties of a process
  - space, time, and energy required by “a program in execution”

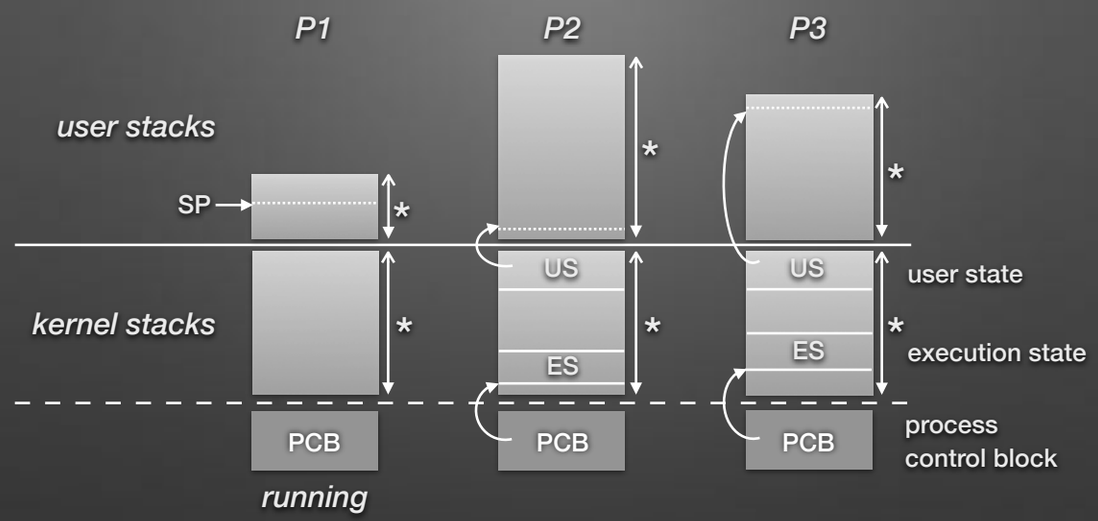


So custom-made operating systems would be ideal, but without reinventing the wheel every time.

In animation order:
1. For this, an operating system should be understood as a program family. Single family members provide customised solutions in relation to a specific use case, while the whole family offers a bunch of solutions to various use cases.
2. The family members have more in common than expected, they are the result of intensive reuse and adaptation of existing programs and modules, respectively.
3. Each of it, however, not only provides a specific subset of system functions but is also characterised by certain non-functional properties.
4. This approach looks easy at first glance. However, the trick is in the detail, especially cross-cutting concerns are challenging.

# Stack space

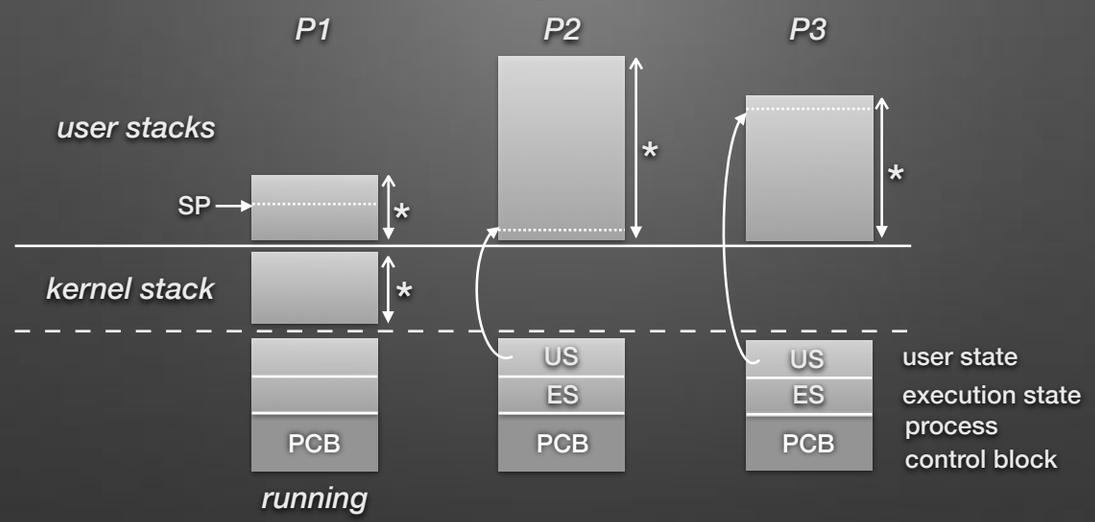
*worst-case stack usage




process-based operating-system kernel

# Stack space contd.

*worst-case stack usage




event-based operating-system kernel

# Prediction of stack usage

program characteristic

processor characteristic

- subroutine nesting: MAX(call graph)
- interrupt service routines: MAX(interrupt priority level)
  - edge-triggered: MIN(inter-arrival time) vs. BCET*
  - level-triggered
    - re-entrant: MIN(interrupt receipt latency) vs. BCET*

environment (external process)

system characteristic

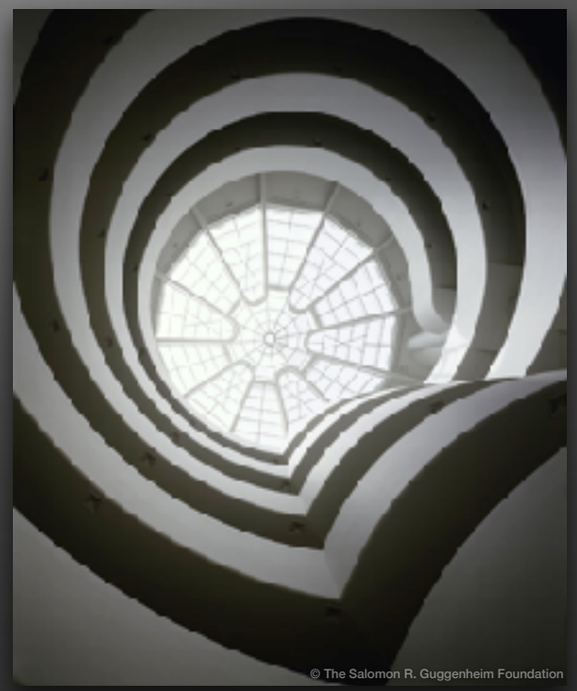
*best-case execution time

# Architectural concerns

far in excess of a certain memory footprint:

- degree of pseudo parallelism through preemption
- kind of penetration or anchoring of concurrency

☛ *Multitasking*



# Preemption

one stack per instance

process-based: *lower latency*

- after any machine instruction, only in case of non-blocking synchronisation
- at selected preemption points, otherwise

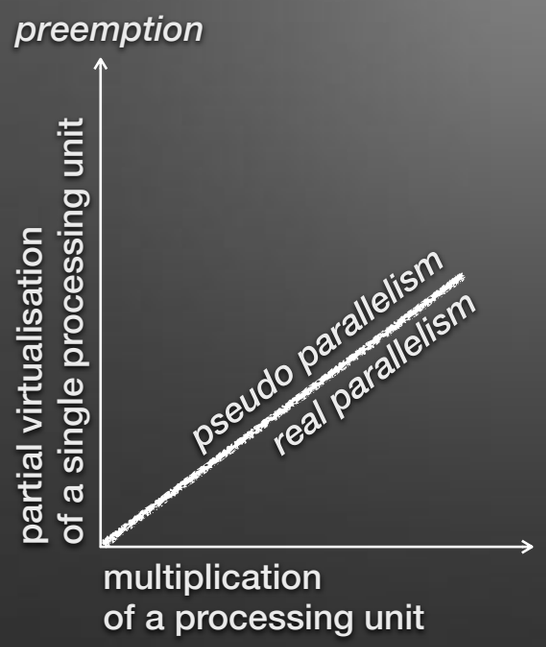
one stack per kernel

event-based: *higher latency*

- at selected preemption points, continuations assumed
- never in kernel space, otherwise

*depending on the level of abstraction*

# Concurrency





*depending on the kind of rootedness*

# Concurrency contd.

process-based: *deep parallelism*

- the kernel is established by a non-sequential program
- partial virtualisation operates above instruction set architecture (ISA) level

event-based: *flat parallelism*

- the kernel is established by a “semi-sequential” program
- partial virtualisation operates above kernel level, only


*depending on the kind of rootedness*

# Timing

Scheduling interference

# Sharing

process scheduling:

- sequencing of actions
- at intended/actual moment of resource provisioning

☛ strong estimates

process synchronisation:

- sequencing of actions
- at intended/actual moment of resource access

☛ factual knowledge

*depending on the type of resource*

# General semaphore

```
procedure acquire(sema)
    sema.load ← sema.load - 1
    if sema.load < 0 then
        enlist(self, sema.list)
        block
    end if
end procedure
```

wrong reading

lost wake-up

priority violation

*the tip of the iceberg…*

```
procedure release(sema)
    sema.load ← sema.load + 1
    if sema.load ≤ 0 then
        next ← delist(sema.list)
        ready(next)
    end if
end procedure
```

# Relinquish processor

```
procedure block
    self.trim ← BLOCKED
    next ← quest(R2R)
    if next = self then
        gauge(self)
    else
        seize(next)
    end if
end procedure
```

priority falsification

double personality

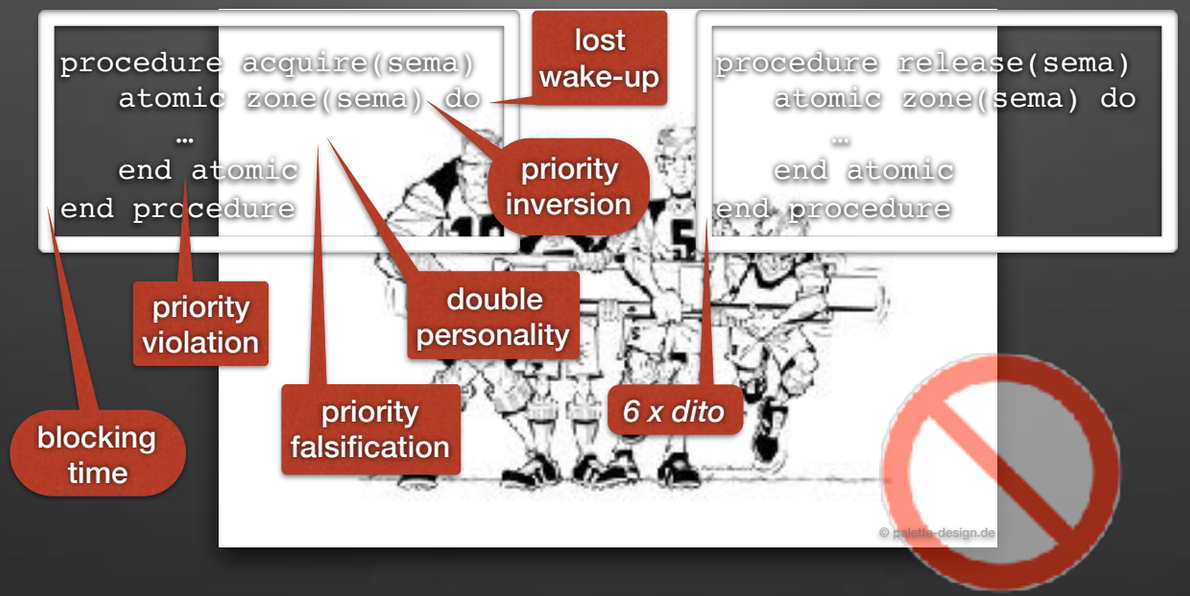
```
function quest(pool)
    repeat
        next ← elect(pool)
        if next = 0 then
            halt
        end if
    until next ≠ 0
    return next
end function
```

lost wake-up

*trials and tribulations of switching processes…*

# Broad brush approach

- multilateral blocking synchronisation: *mutual exclusion*



# Non-blocking synchronised

```
procedure acquire(sema)
   enlist(self, sema.list)
   if FAA(sema.load, -1) ≤ 0 then
      block
   else
      unlist(self)
   end if
end procedure
```

*no lost wake-up!*

double personality

priority falsification

priority violation

*no blocking time!*

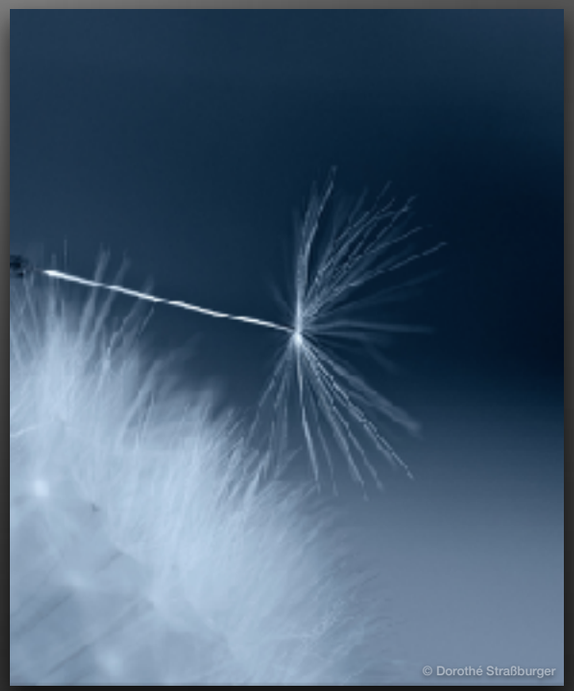
*no priority inversion!*

```
procedure release(sema)
   if FAA(sema.load, 1) < 0 then
      next ← delist(sema.list)
      ready(next)
   end if
end procedure
```

# Ease of being

processes may appear ‘closer’ then they are

- double personality
  - logically blocked or ready
  - but physically running
  - ➡ known from *idle loop*, e.g.
- priority violation
  - unlike queuing disciplines
  - ➡ follow *scheduling order*



# Reconsider decisions

a resuming process has to incur liability to back-pedal

- priority falsification
  - process dispatching never happens indivisible!
  - low-urgent process can lag medium-urgent process
- raise a ‘process obligation’
  - check for a pending higher urgent process
  - if any, relinquish processor

# Non-blocking sync. revisited

```
procedure acquire(sema)
   enlist(self, sema.list)
   if FAA(sema.load, -1) ≤ 0 then
      block
   else
      unlist(self)
   end if
end procedure
```

*no lost wake-up!*

double personality ✓

priority falsification ✓

priority violation ✓

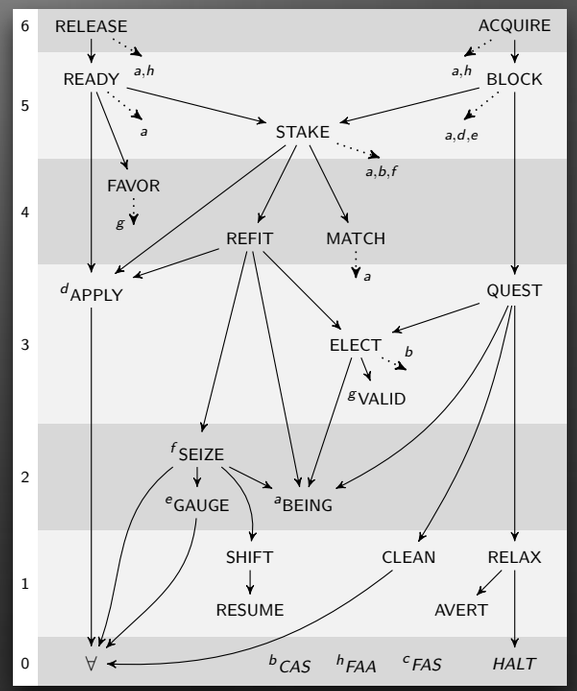
*no blocking time!*

*no priority inversion!*

```
procedure release(sema)
   if FAA(sema.load, 1) < 0 then
      next ← delist(sema.list)
      ready(next)
   end if
end procedure
```

# Non-blocking & wait-free

- a glimpse under the surface:
  - 6 resource assignment
  - 5 basic process control
  - 4 priority control
  - 3 process scheduling
  - 2 process dispatching
  - 1 processor control
  - 0 elementary operations




> One should never despair of impossible things, never of arduous ones. (Goethe)

One should never despair of impossible things, never of arduous ones. (Goethe)

# Interference

- almost prevented when using non-blocking synchronisation
  - atomic read-modify-write machine instructions
  - cooperation with hardware
- non-trivial remaining issue is data-structure handling
  - prevent bad alignment and false sharing
  - cache (d)effects

# Interference contd.

process synchronisation is not the only problem area:

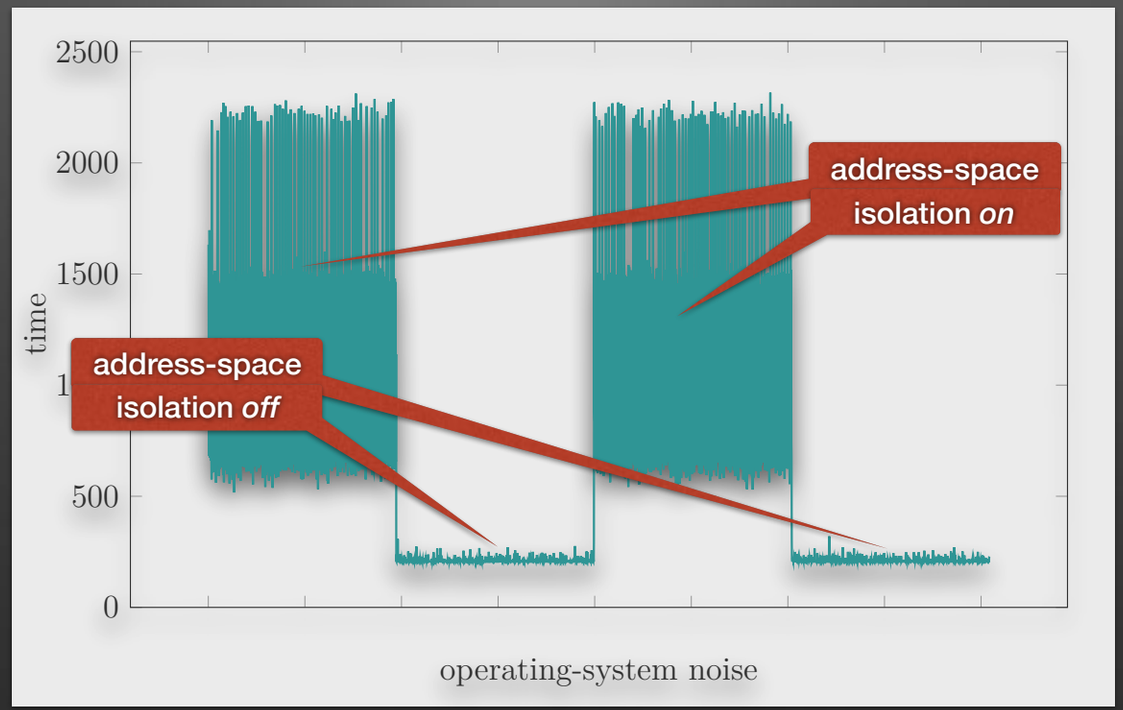
- it is always simply a means to an end
  - coordination
  - communication
  - integrity preservation
  - consistency safekeeping
- operating-system noise breeds trials and tribulations

# Consistency safekeeping

fresh from the operating-system kitchen, just one example:

- page-table maintenance for multi-core systems
  - shared memory
  - replicated page descriptors
  - translation lookaside buffer (TLB) handling
- inter-processor interrupt (IPI), the root of all evil

# Inter-processor interrupts



# Address-space isolation ‘on demand’

**“Some users may require only a subset of the services or features that other users need. These ‘less demanding’ users may demand that they not be forced to pay for the resources consumed by the unneeded features.”**

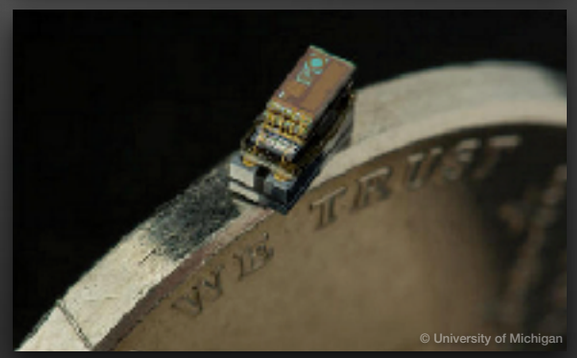
***–David Parnas, 1979****

***Designing Software for Ease of Extension and Contraction, IEEE TSE, vol. SE-5, no. 2**

# Energy

Efficient operation








In animation order:

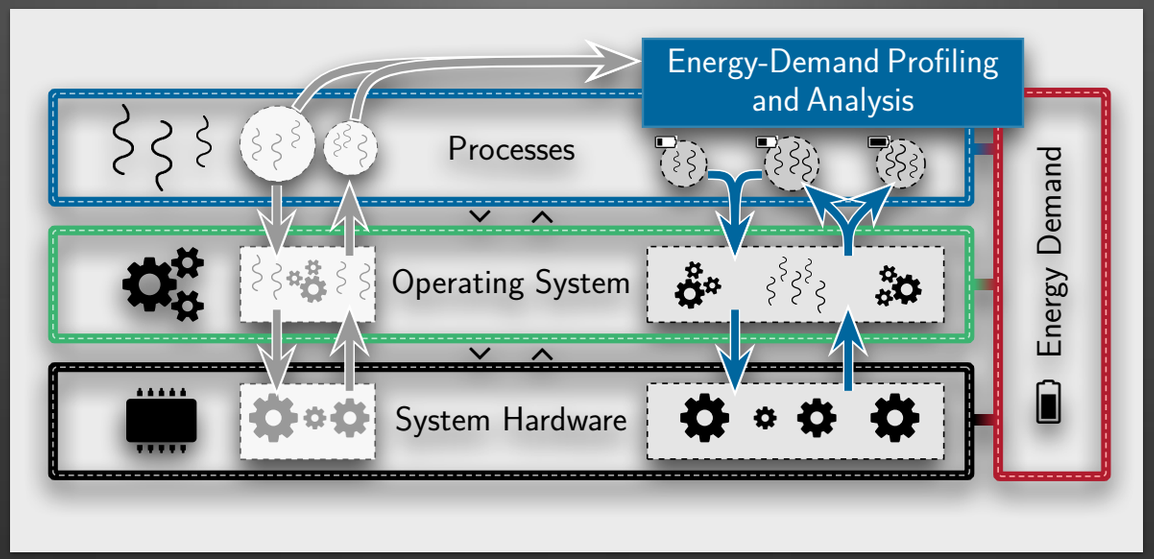
1. Energy has always been a precious and scarce resource, but it has gained more and more esteem only in the last decades.
2. This applies in particular to machine giants for high-performance computing, big-data processing, or Bitcoin production. A single Bitcoin transfer today (2018) needs as much power as a US citizen in a week with 250 kilowatt hours. Or for example Iceland, where Bitcoin mining operations will use around 840 gigawatt hours of electricity to supply the data centres while all the homes together merely spend around 700 gigawatt hours every year.
3. But it also applies to machine dwarves like smart dust, wearable computers, microcontrollers, and especially to the countless devices that make the Internet of Things. Small cattle makes a mess, in other words, the energy needs of those small-scale computers in their entirety is in no way inferior to the supercomputer.

# Prediction protects against nasty surprise

hardware converts energy — but software determines how much

- estimate software-induced energy demand
  - basic-block level
    - ➡ static program analysis
- useful quantification requires suitable hardware models
  - where to take, if not steal?
    - ➡ machine learning

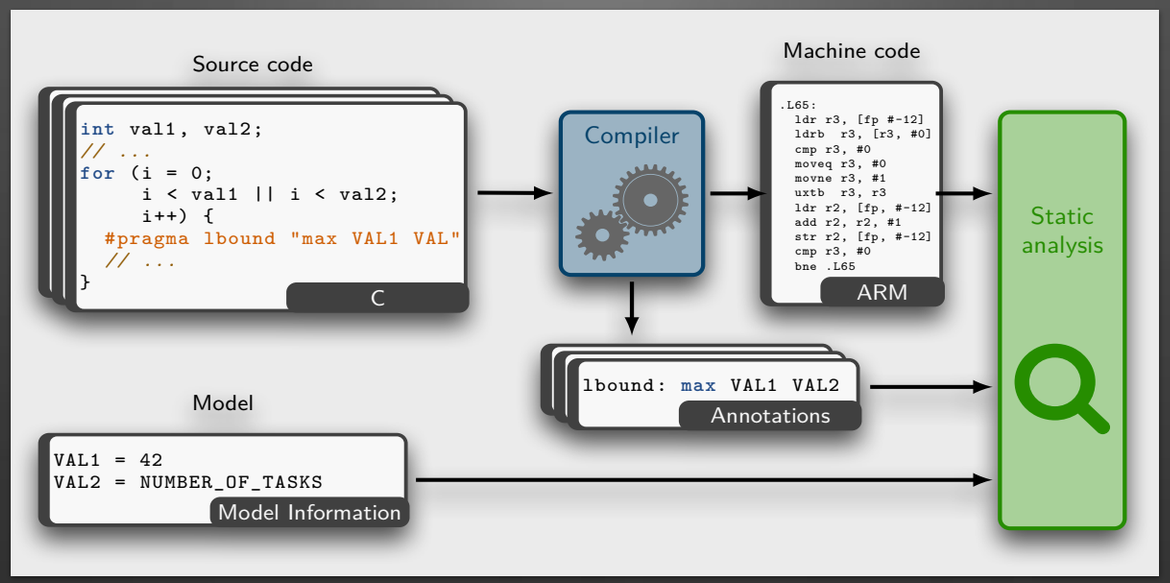
# Energy-aware programming




Behind energy-aware programming is a multi-phase and cross-cutting approach for the resource-saving operation of a computing system as to energy need and reserve. It is based on:

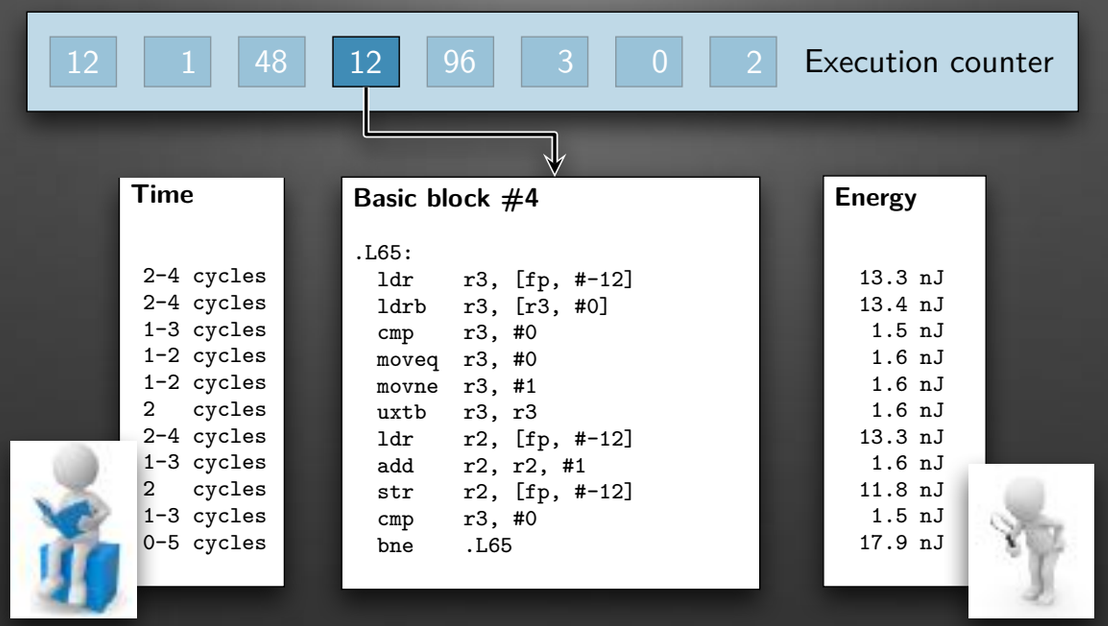
1. static and dynamic program analysis to determine the energy demand of selected processes,
2. a tooling infrastructure for the development of proactive energy-aware programs and multi-variant energy demand analysis,
3. an operating-system executive that aims at reducing the energy need of processes in a cross-layer manner, and
4. an integrated energy measurement method supplemented by a suitable auxiliary device for lossless demand recording.

# Accumulate knowledge




At the beginning is the as far as possible automatic extraction of knowledge for the expected energy demand of the programs intended for execution. This step is similar to the WCET analysis common for real-time systems, but has its focus on the worst-case energy demand, not execution time, of the examined programs. Incomplete knowledge from source-code analysis, such as loop bounds or specific system parameters, is completed with application- as well as configuration-dependent model information. This way, the unknowns declared by source-code annotations are resolved in the subsequent static program analysis step.

# Quantify needs

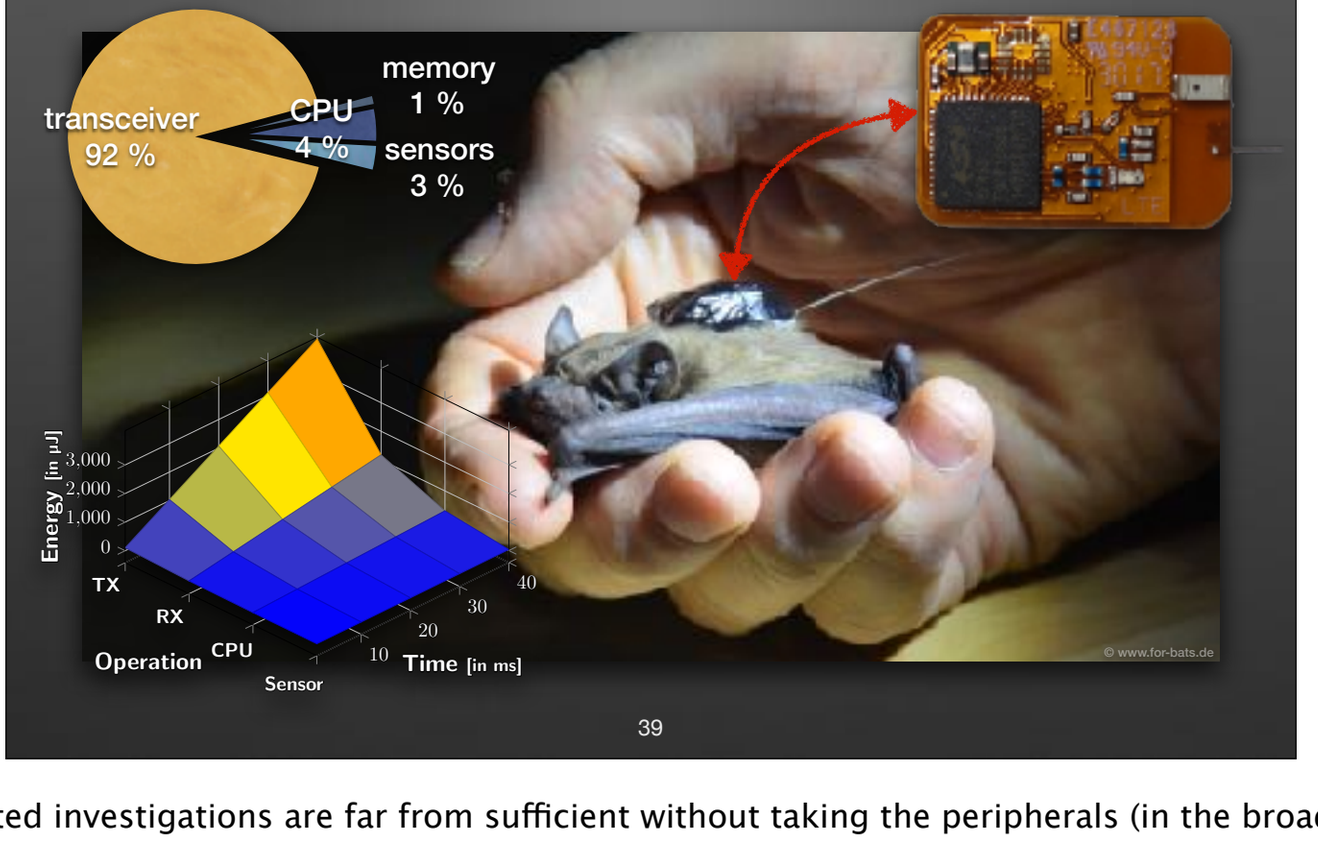





In animation order:

1. Energy needs are recorded, estimated, and measured, respectively, at basic-block level. The execution number of each basic block is determined by (static/dynamic) program analysis.
2. As far as timing is concerned, the corridor for the execution time is extrapolated using the processor cycles of each instruction.
3. For the derived unit of energy, similar is been done to obtain the parameter for a particular basic block.
4. However, while the processor cycles expected per instruction are obtained simply by reading data sheets, the corresponding energy need in nanojoules is to be determined by elaborate measuring.

# Involve peripherals






But all these on internal things oriented investigations are far from sufficient without taking the peripherals (in the broadest sense) as to the likewise given use case into account.

Further explanation in animation order.

# Where the shoe pinches

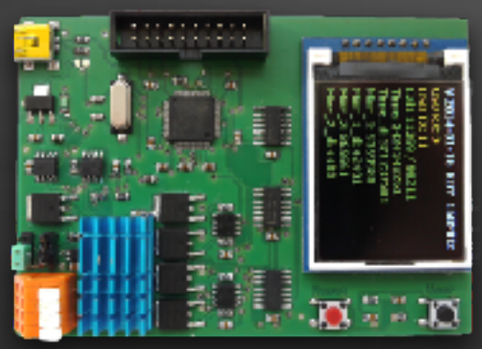
prediction stands and falls with demand details

- internal characteristic
  - CPU, main memory, …
- external characteristic
  - peripherals

no hands no biscuits — without energy model no estimate

- measure by hand
- machine learning

# Empirical data acquisition

www4.cs.fau.de/Research/MeasureAlot

1. measure power demand at basic-block level
2. automate process to create a representative data set
3. generate energy model using a deep neural network

# Summary

# Predictability…

*… is always subject to the underlying assumptions being made and relates to the dimensions along which real-time systems can be categorized**

- deadlines (granularity, strictness), laxities for tasks
- reliability requirements
- system size, interaction, environmental characteristics

- design for predictability is an overarching aspect that crosscuts the whole computing system

***J. A. Stankovic, K. Ramamritham, What is Predictability for Real-Time Systems?, 1990**

# Recommended reading

- A. A. Fröhlich, Application-Oriented Operating Systems, 2001
- G. Drescher and W. Schröder-Preikschat, An Experiment in Wait-Free Synchronisation of Priority-Controlled Simultaneous Processes: Guarded Sections, 2015
- P. Wägemann et al., Worst-Case Energy Consumption Analysis for Energy-Constrained Embedded Systems, 2015
- T. Hönig, Proactive Energy-Aware Computing, 2017
- P. Wägemann et al., Operating Energy-Neutral Real-Time Systems, 2018

# Acknowledgement