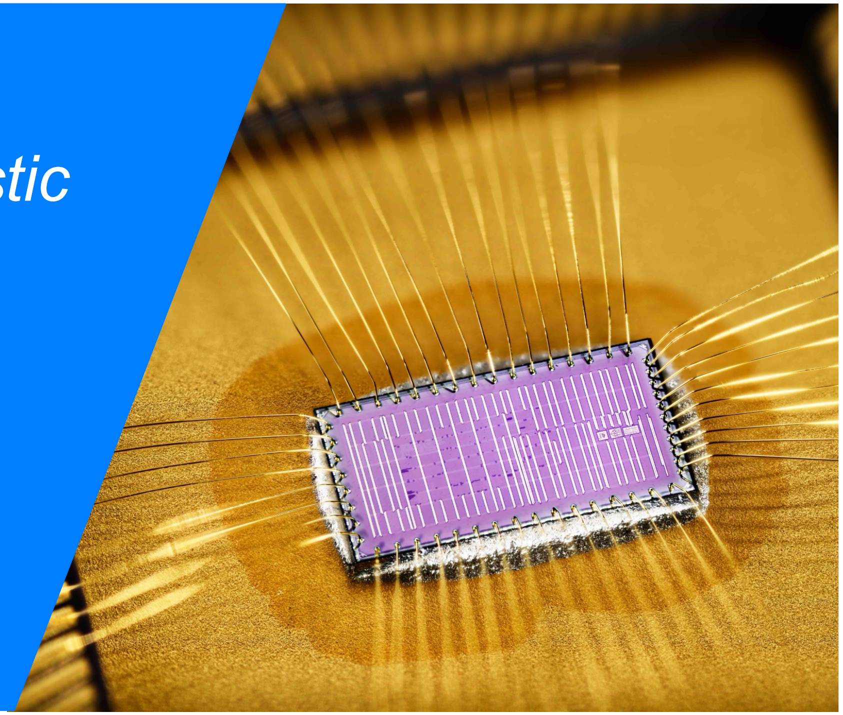
CompSOC: A composable, predictable, and deterministic multi-core platform and its modelling

Kees Goossens
Andrew Nelson, Martijn Koedam,
Dip Goswami, Twan Basten, Marc Geilen,
and many others



Kees Goossens <k.g.w.goossens@tue.nl>
Electronic Systems Group
Electrical Engineering Faculty

HAL4SDV
Systems Safety Security Software

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

goals

- research
  - bridge the gap between theory and practice by
    - building systems that are a priori analysable rather than post hoc try to analyse COTS
    - compare their cost:performance

- for this seminar
  - see the state of the art on synchronous languages
  - build awareness of our real-time deterministic multicore platform
  - find out if & how it can be a target for synchronous languages & similar design methodologies

overview of CompSOC

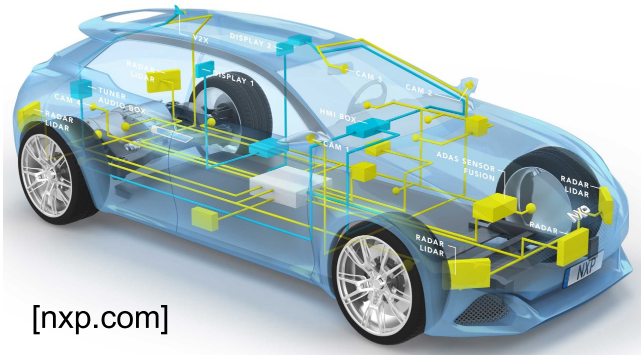
- goals
- concepts
- platform
- MOCs & analysis
- design flow
- examples

- I'm sure there will no time, but... component model

embedded systems

- interaction with physical world → real time, possibly safety critical
- cost & resource constrained (volume, power, energy, ...)
- to a large degree, application specific → designing must be efficient
- multiple applications, often from different providers → integration must be efficient

[abb.com]



[nxp.com]

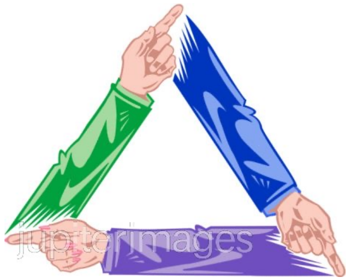
Apple vision [ifixit.com]

applications

- applications have different characteristics
  - hard, soft, non real-time, adaptive, ...
  - different models of computation & verification approaches

- applications are developed & tested independently
- monolithic re-verification after integration

- blame game
- no guarantee of convergence



CompSOC goals

- reduce system design and verification effort
  - including provable real-time performance where needed

- independent design, verification, and deployment per application

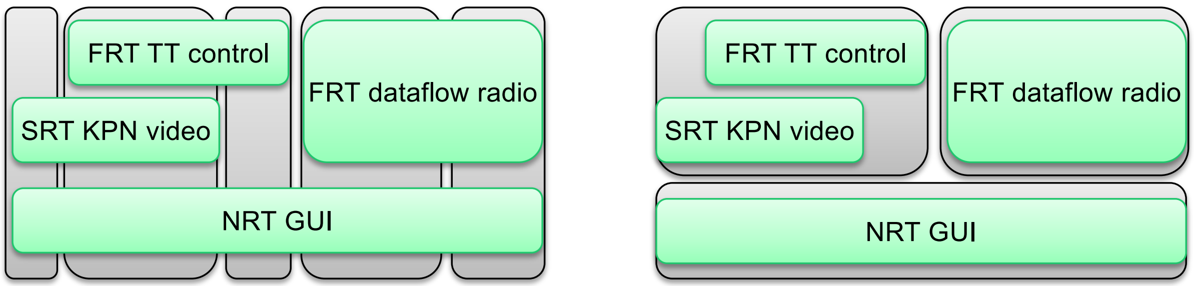
- two complexity-reducing techniques
  - composability
  - predictability (incl. determinism)

- ingredients
  - hardware — multiprocessor
  - middleware — microkernel, middleware, reconfiguration, etc.
  - model(s) of computation — e.g. SDF for real-time applications, synchronous, TT
  - design flow — synthesis & mapping of hardware & software

composability

- virtual execution platform (VEP) per application

- design, verify, deploy in a virtual execution platform
- no interference
  - when integrating, executing, or switching use cases



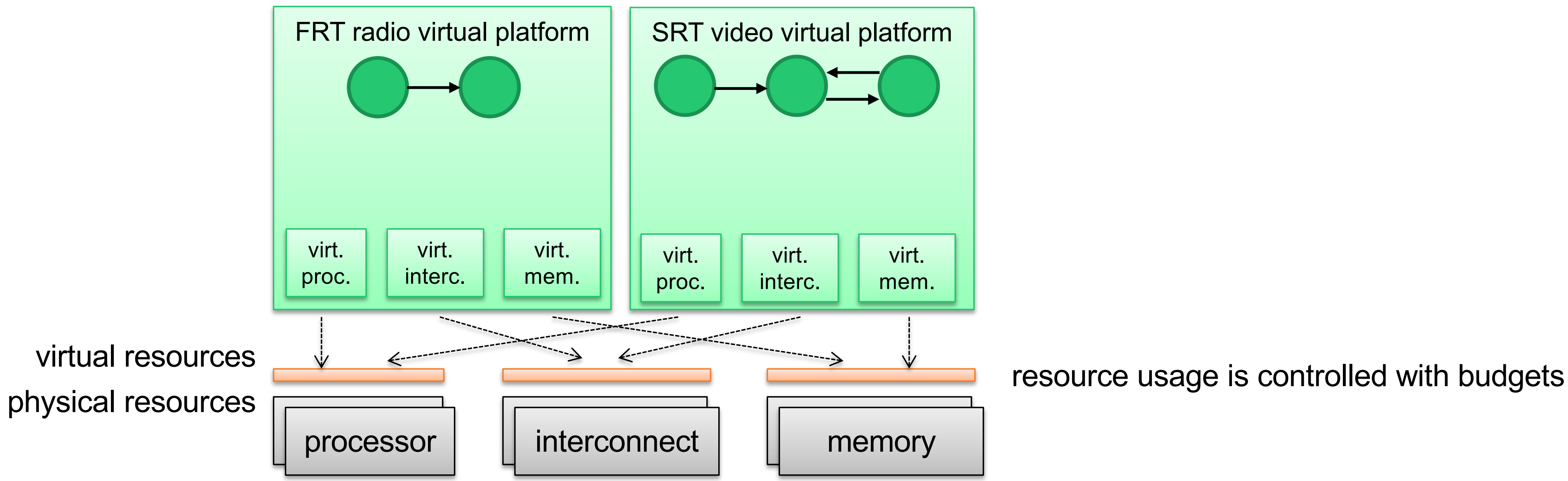
N! combinations & interleavings ➔ N virtual platforms

inter-dependent state ➔ independent states

$2^S \times 2^S \times 2^S \times 2^S$ ➔ $2^S + 2^S + 2^S + 2^S$

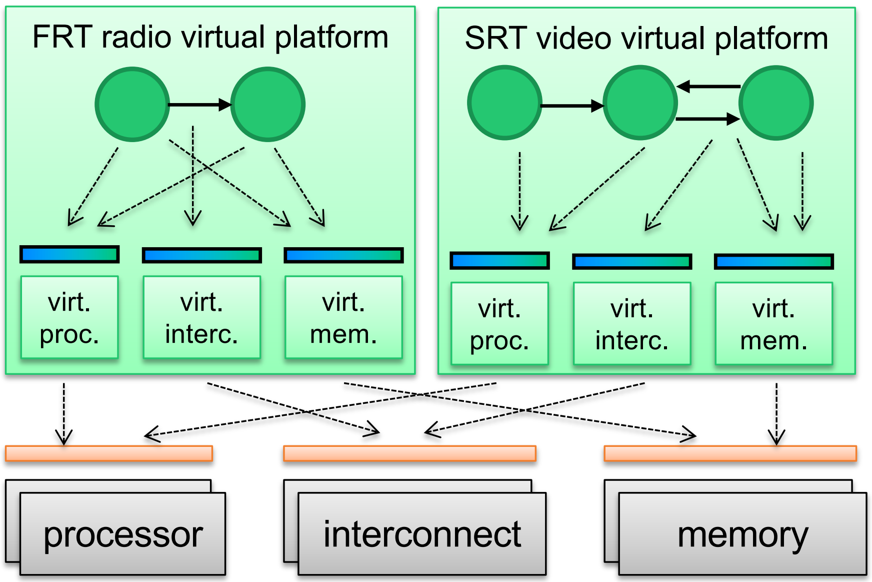
composability

- virtual execution platform consists of virtual resources
- a virtual resource is defined by its space, time, & energy budget



composability & predictability

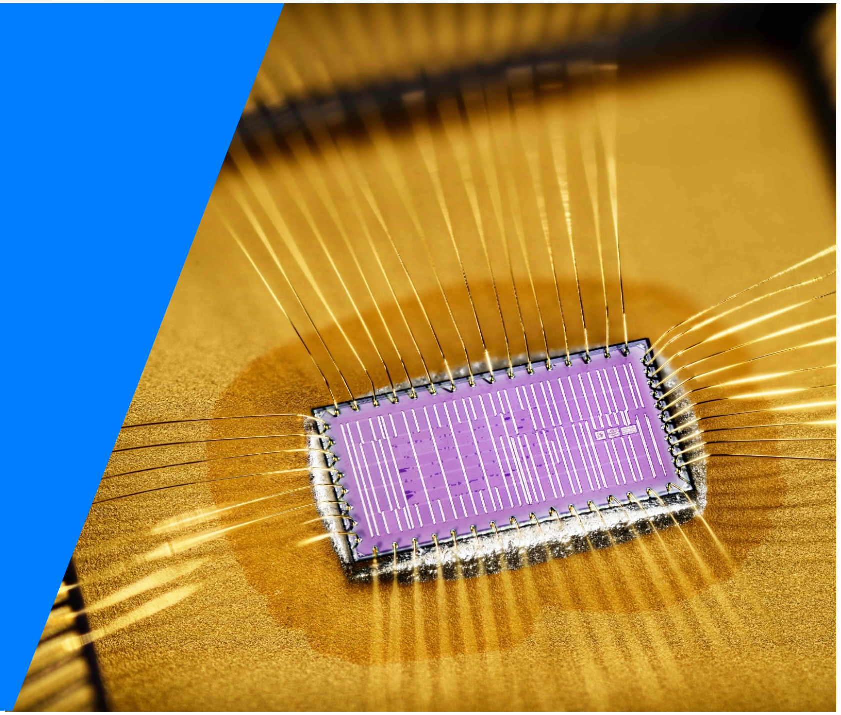
- each application can have its own
  - programming model, scheduling, power management
  - e.g. DF, KPN, sync/TT, or best-effort using distributed shared memory
- for real-time applications use appropriate MOC, as well as predictable schedulers



intra-application scheduling
& power management

inter-application scheduling

CompSOC platform

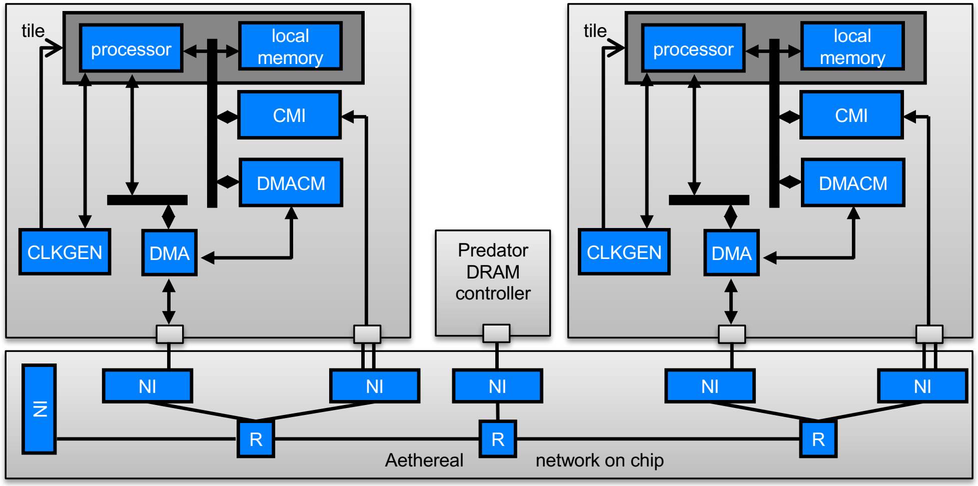


Kees Goossens <k.g.w.goossens@tue.nl>
Electronic Systems Group
Electrical Engineering Faculty

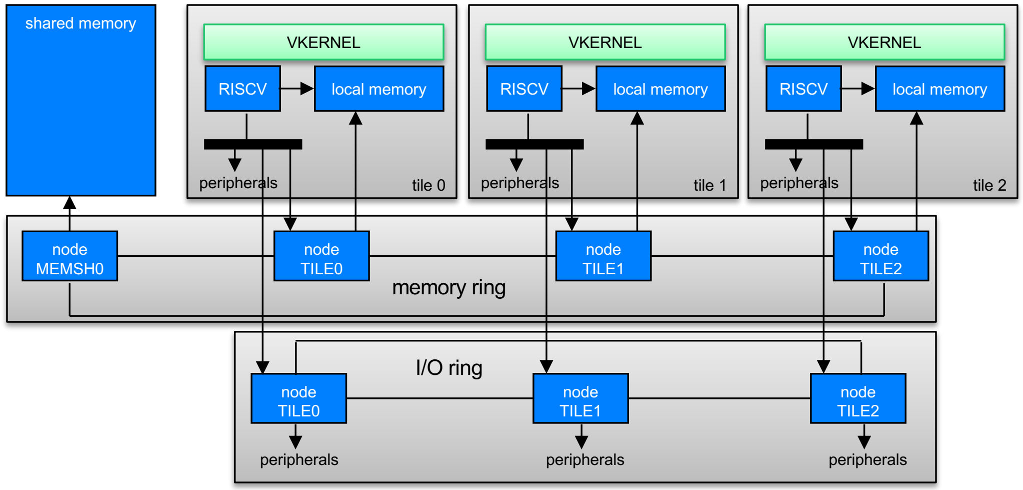
EINDHOVEN UNIVERSITY OF TECHNOLOGY

CompSOC platform template

- processors: RISC-V, microBlaze, ARM M0; MMU
- memory: SRAM tiles, Predator DRAM controller
- communication: Aethereal network on chip; token ring; DMA
- software: two-level scheduling, microkernel, middleware, resource management / reconfiguration



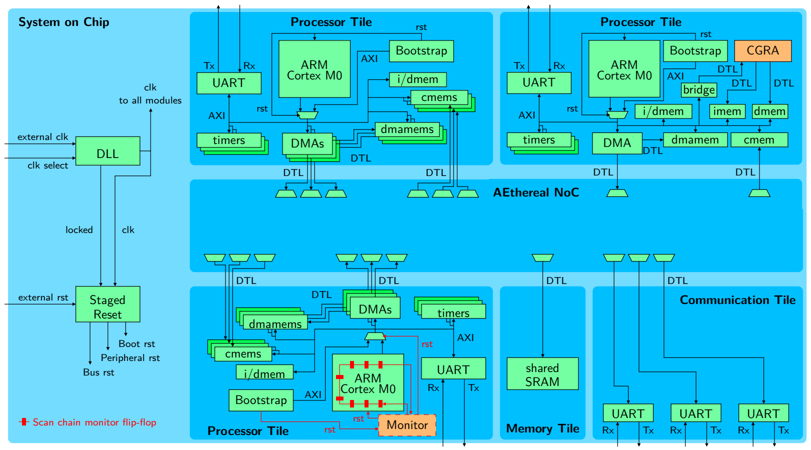
first CompSOC [TECS'09]

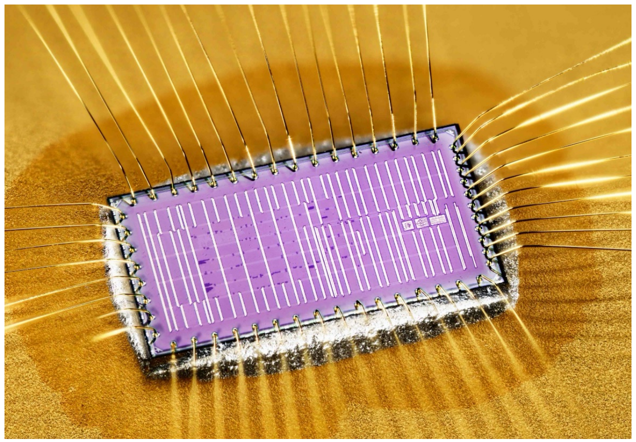


latest CompSOC

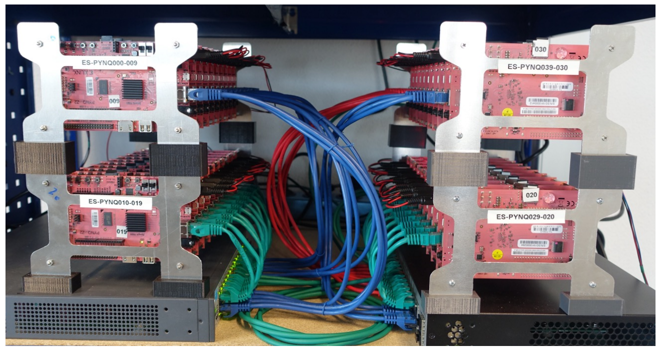
CompSOC ASIC 2019

- 3 Arm M0 tiles
  - one with resiliency features
  - one with Coarse Grain Reconfigurable Array (CGRA)
- SRAM tile
- I/O tile 3x UART
- 2x2 NOC

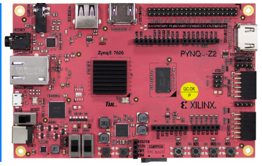


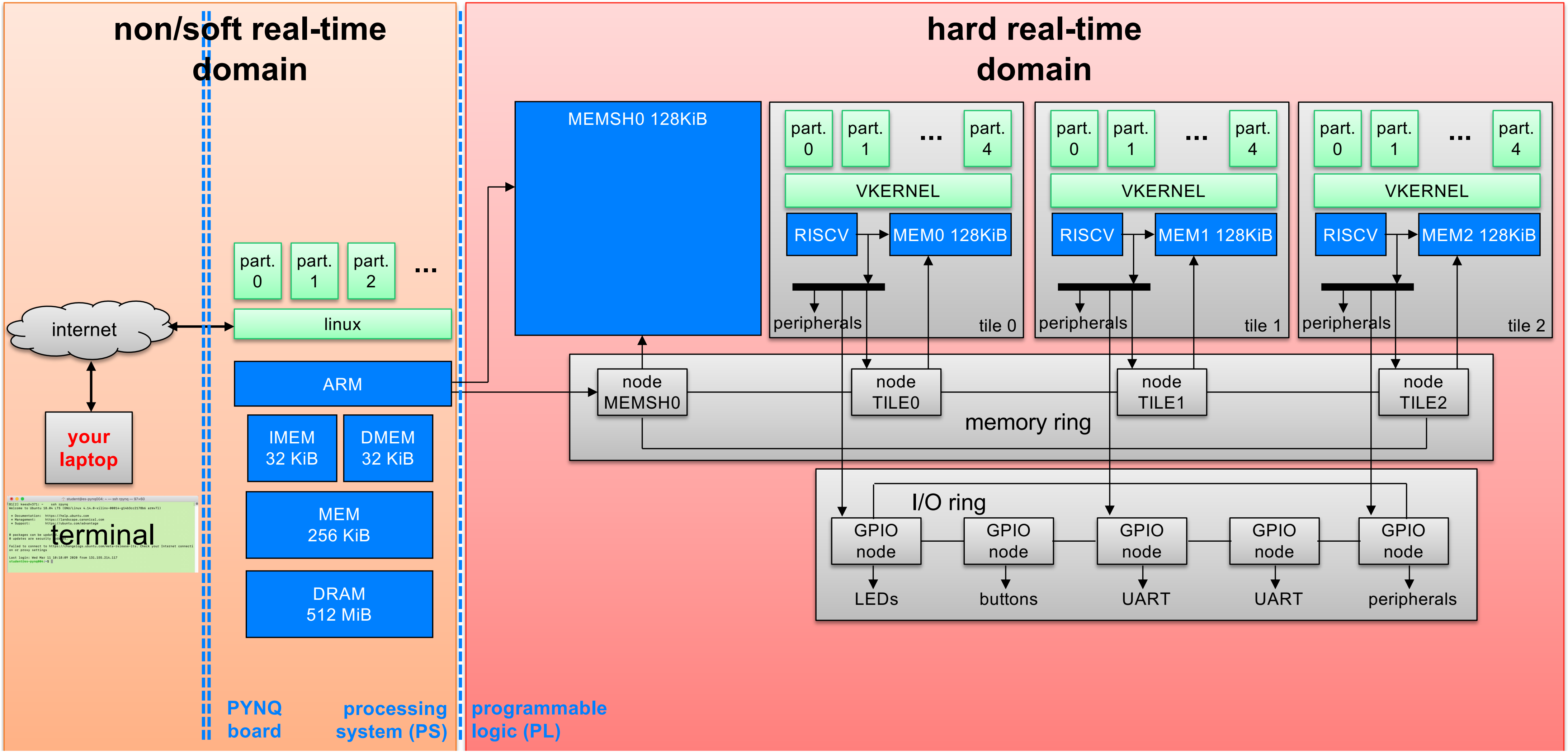


various FPGA prototypes



latest CompSOC FPGA implementation by Verintec





command-line system management on Linux on Arm

- specify budgets for applications, e.g.

```
on tile 0 partition 1 has 8KiB stack in 32KiB memory starting at 32KiB
on tile 0 partition 1 shares 1KiB private memory
in shared memory memshared0 shares 1KiB private memory starting at 60KIB
on tile 0 partition 1 has 10000 cycles of 43000 starting at  7000
```

- dynamic load, start, suspend, resume, stop of applications on the CompSOC instance on the FPGA
  - without any impact on other running applications
  - at specified cycle-accurate point in time

- state saving & restoring of applications
  - e.g. for back up, duplication, migration

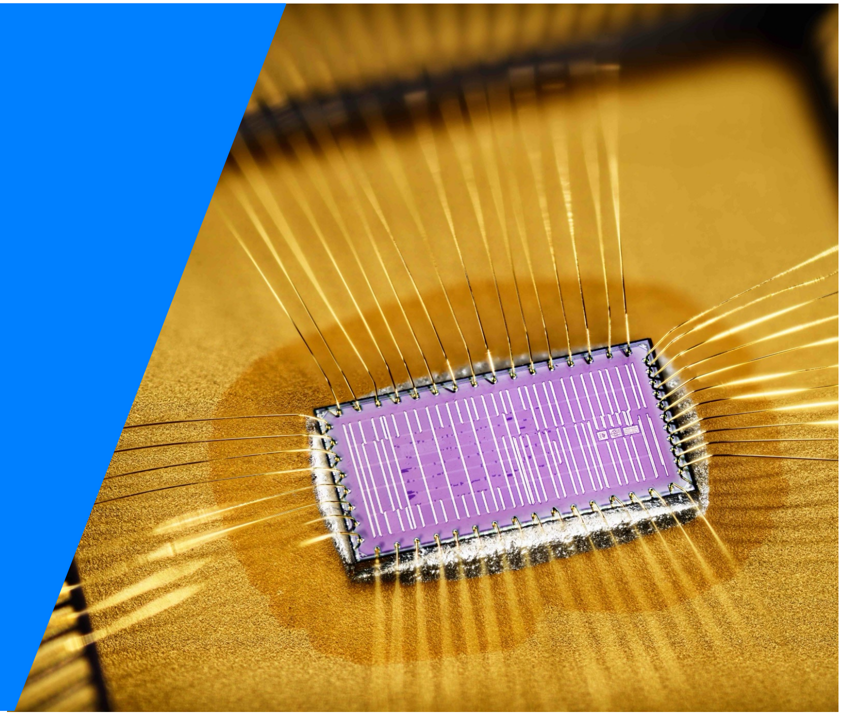
CompSOC platform template

avoiding, e.g.

- hidden state in processors (e.g. branch prediction, pipeline, status flags,
  → side channel attacks such as Spectre/Meltdown)
- imprecise interrupts, power management, etc.
- internal activity, e.g. DRAM auto refresh
- scheduling anomalies in processors, interconnect, memory, schedulers
- unpredictable run-time due to e.g. branch prediction, caching, coherence, DRAM CDC
- blocking & coupled resources (e.g. processor, cache, interconnect, cache, ...; atomic instructions e.g. test & set, fences,)
- RISC-V load reserved/store conditional is not composable and can cause starvation

composable	→ space & time partitioned with budgets
	→ zero cycles of interference
	→ cycle-accurately identical running bare metal or in a VEP
	→ stateless application switching
predictable	→ budget schedulers
	→ precise & accurate WCET & WCRT
deterministic	→ cycle-accurate control & replay

Applications



Kees Goossens <k.g.w.goossens@tue.nl>
Electronic Systems Group
Electrical Engineering Faculty

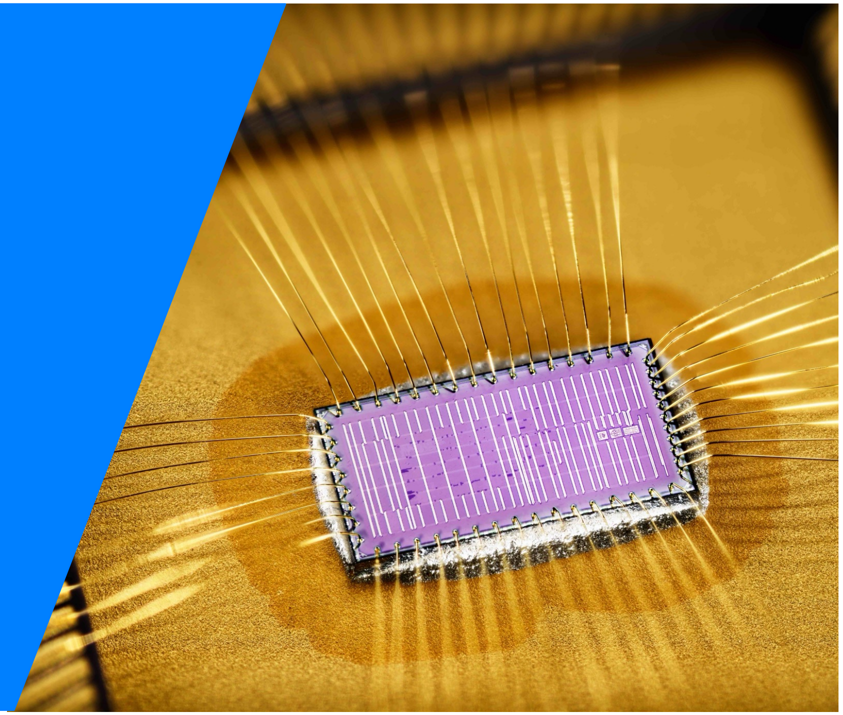
EINDHOVEN UNIVERSITY OF TECHNOLOGY

applications

- an application consists of one or more executables running in a VEP
- depending on the application requirements
  – decide on a MOC
    - real time → SDF, synchronous, TT
    - (value) deterministic → KPN
    - best effort, adaptive → C, ...
  – implement the actors/tasks/threads/...
  – implement the application graph
  – determine application WCRT, if required
  – run

– may need to update VEP
– integration with other apps → zero effort!

Example: dataflow



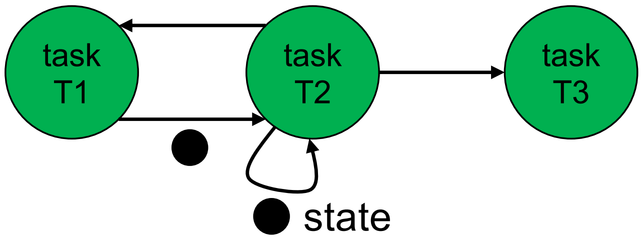
Kees Goossens <k.g.w.goossens@tue.nl>
Electronic Systems Group
Electrical Engineering Faculty

EINDHOVEN UNIVERSITY OF TECHNOLOGY

dataflow (SDF, MRDF, SADF)

- graph of stateless actors with FIFO buffers between them
- blocking read of all input tokens, then compute output tokens, then blocking write of all output tokens
- token production/consumption does not depend on the value of the tokens

- tasks synchronise on data (tokens), not on time
- although static schedule is also possible

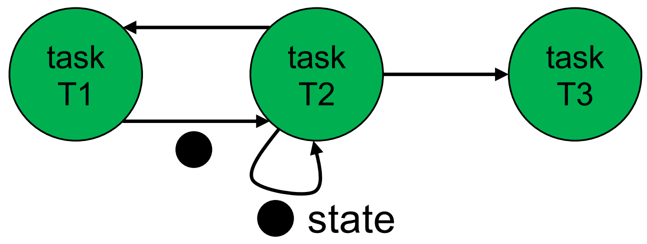



```
do {
  // read input tokens
  while (!input_token){}
  load input_token;

  // function to compute output tokens
  (state_token, output_token) = function(state_token, input_token);

  // write output tokens
  while (!space for output_token){}
  store output_token;
} while (true);
```

dataflow (SDF, MRDF, SADF)

- graph of stateless actors with FIFO buffers between them
- blocking read of all input tokens, then compute output tokens, then blocking write of all output tokens
- token production/consumption does not depend on the value of the tokens

- tasks synchronise on data (tokens), not on time
- although static schedule is also possible

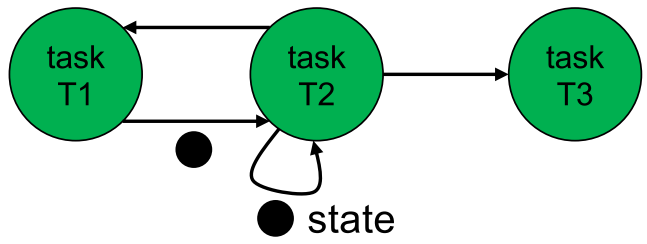



```
do {
  // read input tokens
  while (!input_token){}        ← waiting on others
  load input_token;

  // function to compute output tokens
  (state_token, output_token) = f(state_token, input_token);

  // write output tokens
  while (!space for output_token){}
  store output_token;
} while (true);
```

dataflow (SDF, MRDF, SADF)

- graph of stateless actors with FIFO buffers between them
- blocking read of all input tokens, then compute output tokens, then blocking write of all output tokens
- token production/consumption does not depend on the value of the tokens

- tasks synchronise on data (tokens), not on time
- although static schedule is also possible

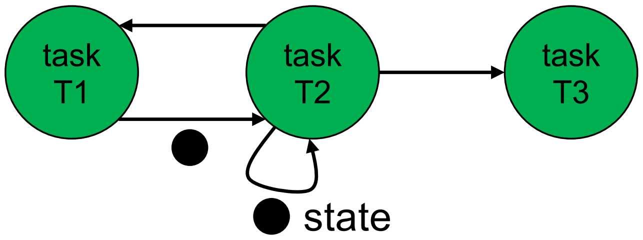



```
do {
  // read input tokens
  while (!input_token){}          ← waiting on others
  load input_token;               ← communication

  // function to compute output tokens
  (state_token, output_token) = f(state_token, input_token);

  // write output tokens
  while (!space for output_token){}
  store output_token;
} while (true);
```

dataflow (SDF, MRDF, SADF)

- graph of stateless actors with FIFO buffers between them
- blocking read of all input tokens, then compute output tokens, then blocking write of all output tokens
- token production/consumption does not depend on the value of the tokens

- tasks synchronise on data (tokens), not on time
- although static schedule is also possible

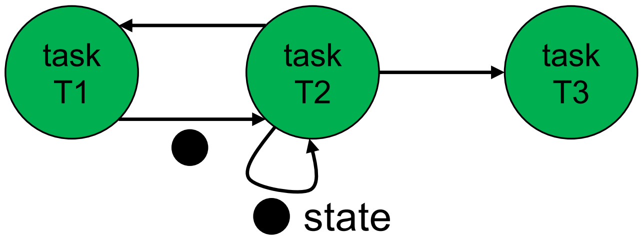



```
do {
  // read input tokens
  while (!input_token){}                                   ← waiting on others
  load input_token;                                        ← communication

  // function to compute output tokens
  (state_token, output_token) = f(state_token, input_token);   ← computation

  // write output tokens
  while (!space for output_token){}
  store output_token;
} while (true);
```

dataflow (SDF, MRDF, SADF)

- graph of stateless actors with FIFO buffers between them
- blocking read of all input tokens, then compute output tokens, then blocking write of all output tokens
- token production/consumption does not depend on the value of the tokens

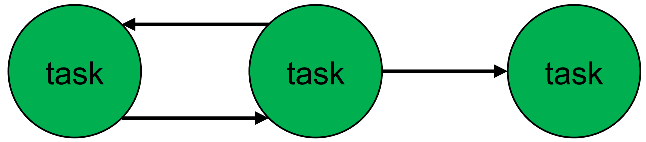
- tasks synchronise on data (tokens), not on time
- although static schedule is also possible



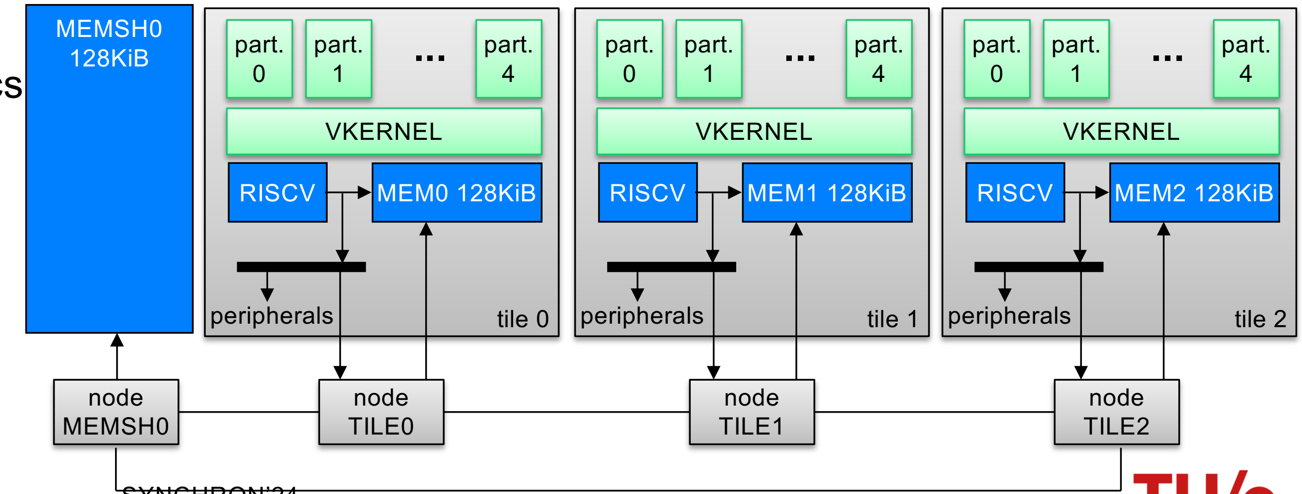

```
do {
  // read input tokens
  while (!input_token){}                                         ← waiting on others
  load input_token;                                              ← communication

  // function to compute output tokens
  (state_token, output_token) = f(state_token, input_token);     ← computation

  // write output tokens
  while (!space for output_token){}                              ← waiting on others
  store output_token;                                            ← communication
} while (true);
```

real time, predictable

- requirements
  - on the application, e.g.
    - while(1) {} → execution time is infinite
    - dataflow graph on the right has a deadlock → throughput = 0
  - on the platform, e.g.
    - wrong schedule(r) can lead to starvation



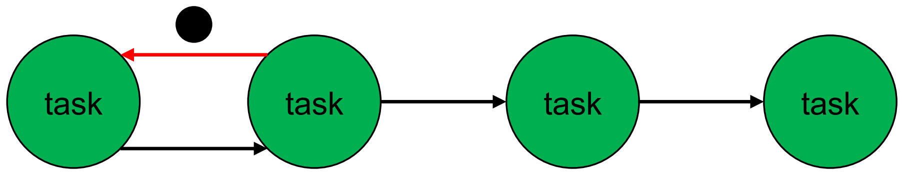
- OK, given
  - a dataflow application
  - a platform with the right characteristics
- how do we compute the throughput of the dataflow application?



dataflow modelling

- we use dataflow to describe

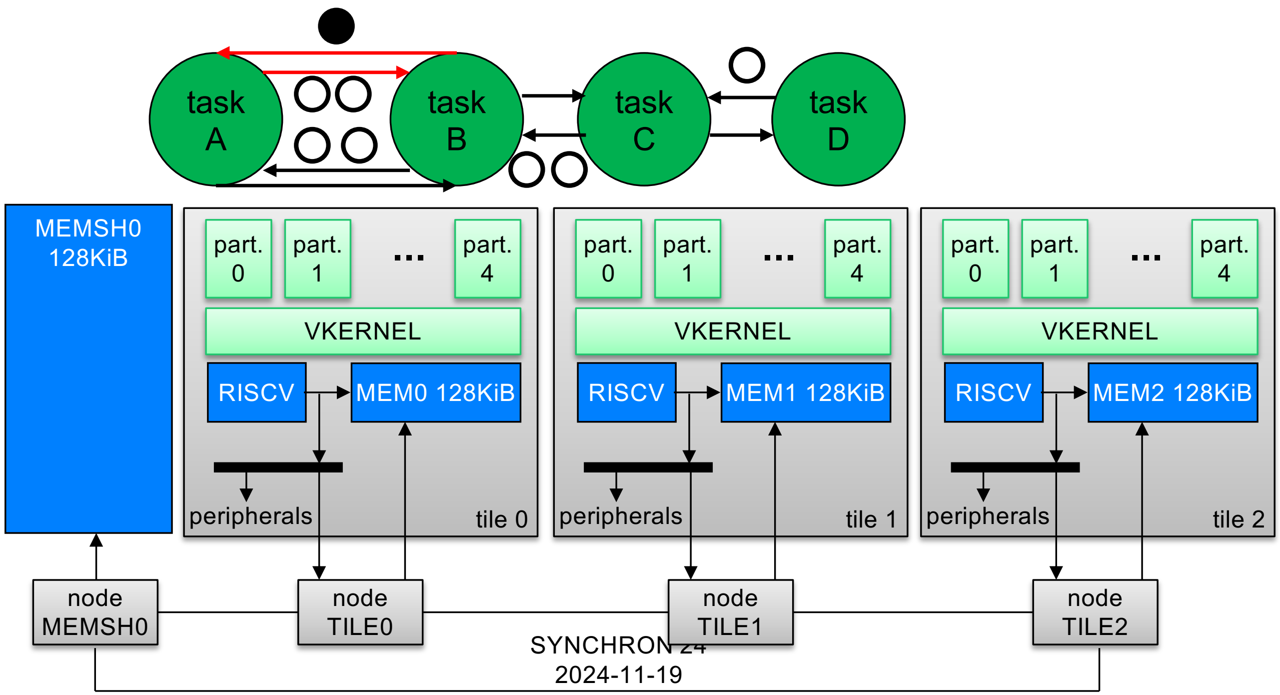
1. the application tasks and their data dependencies
2. auto-concurrency & bounded buffer sizes
3. mapping of actors on non-shared resources → WCET
4. scheduling dependencies
5. resource sharing→ WCRT



- with all those compute the guaranteed (minimum) throughput of an application running on a platform

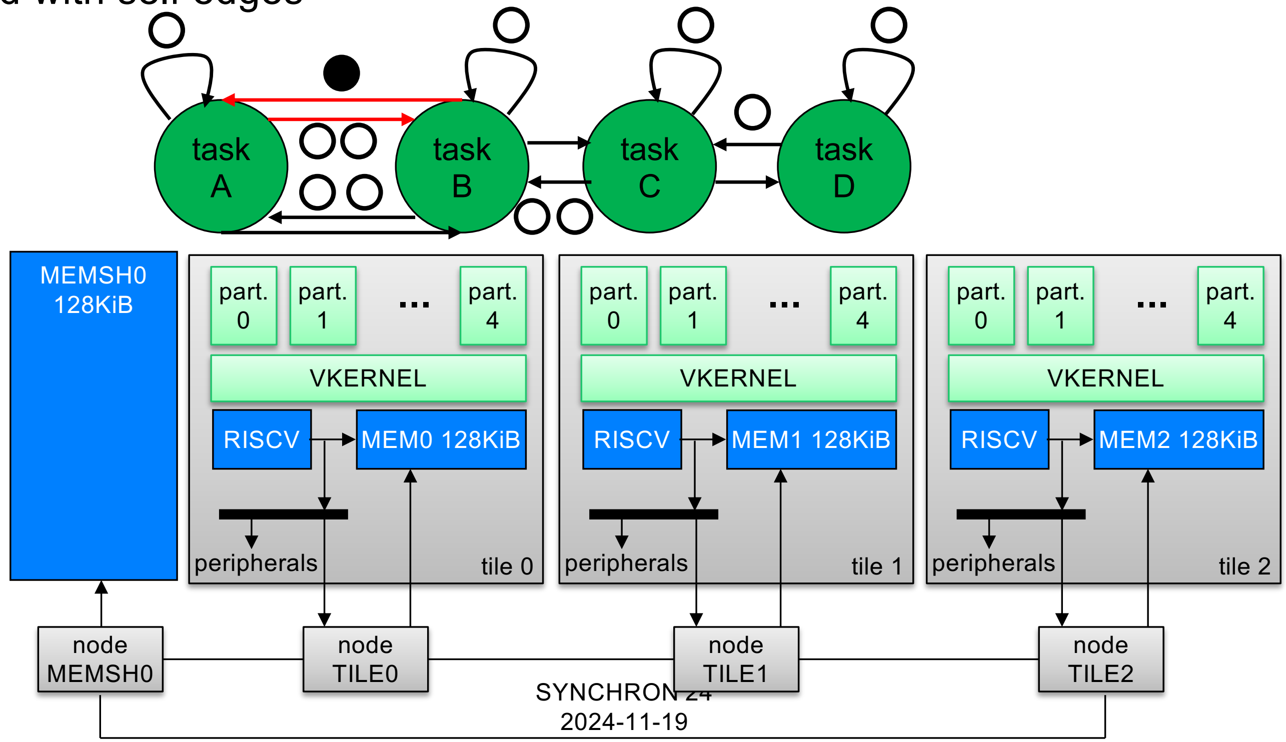
2- modelling finite buffers & auto-concurrency

- since buffers are finite, they must be modelled as such in the dataflow graph
  - all tokens are equal, but for clarity, the newly added tokens are white
  - total number of tokens on the channel pair is the buffer capacity (e.g. 3 for the red channel pair)



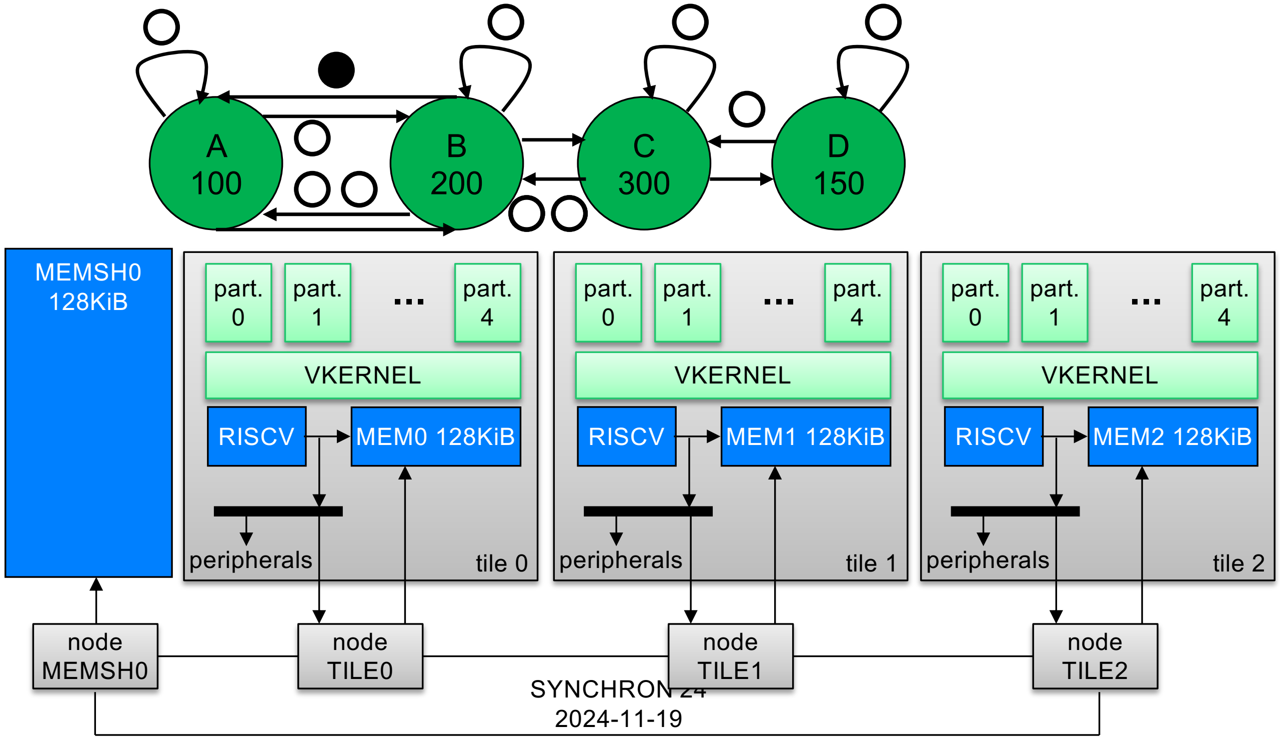
2- modelling finite buffers & auto-concurrency

- since buffers are finite, they must be modelled as such in the dataflow graph
  - all tokens are equal, but for clarity, the newly added tokens are white
  - total number of tokens on the channel pair is the buffer capacity (e.g. 3 for the red channel pair)
- no auto-concurrency: actors must finish before they run again
  - this is modelled with self edges



3 – mapping of actors on non-shared resources → WCET

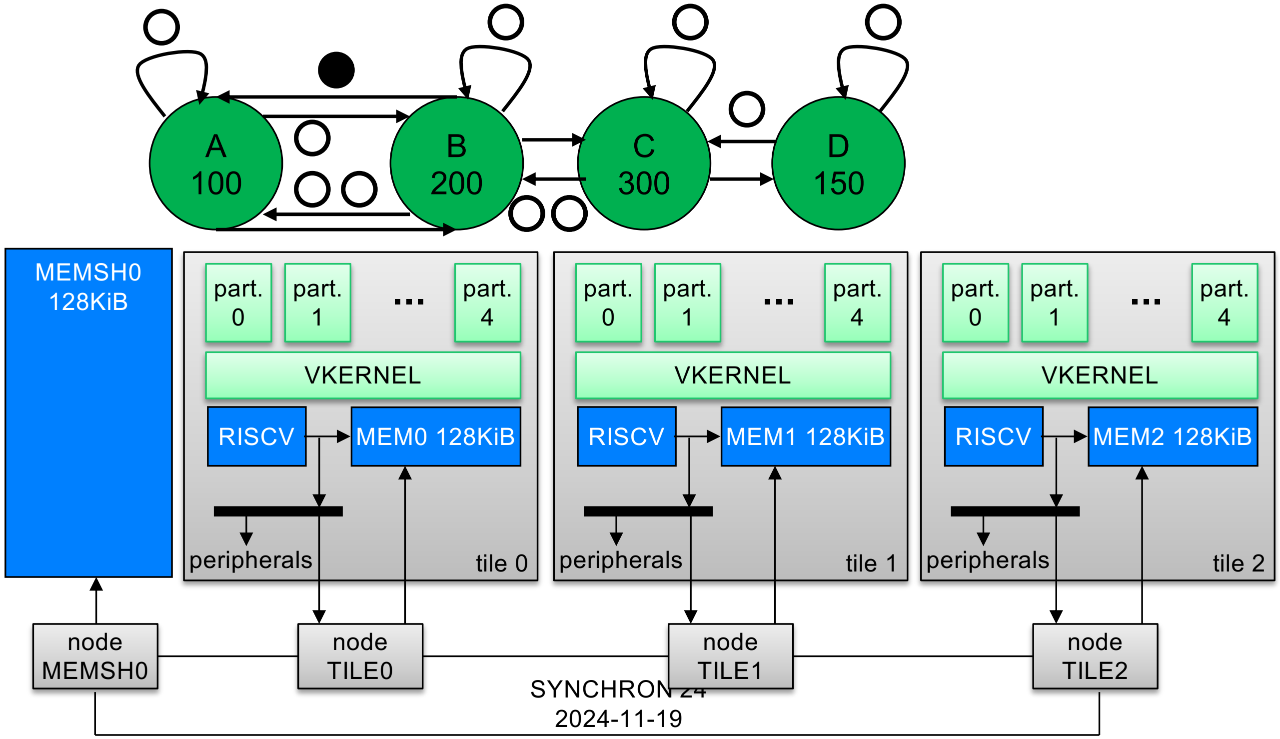
- obtain the WCET of each actor running bare metal on the processor tile
  - e.g. MISRA C with WCET tools (CBMC, moving to LLVM)
  - the actor WCET doesn't include the waiting for tokens



TU/e

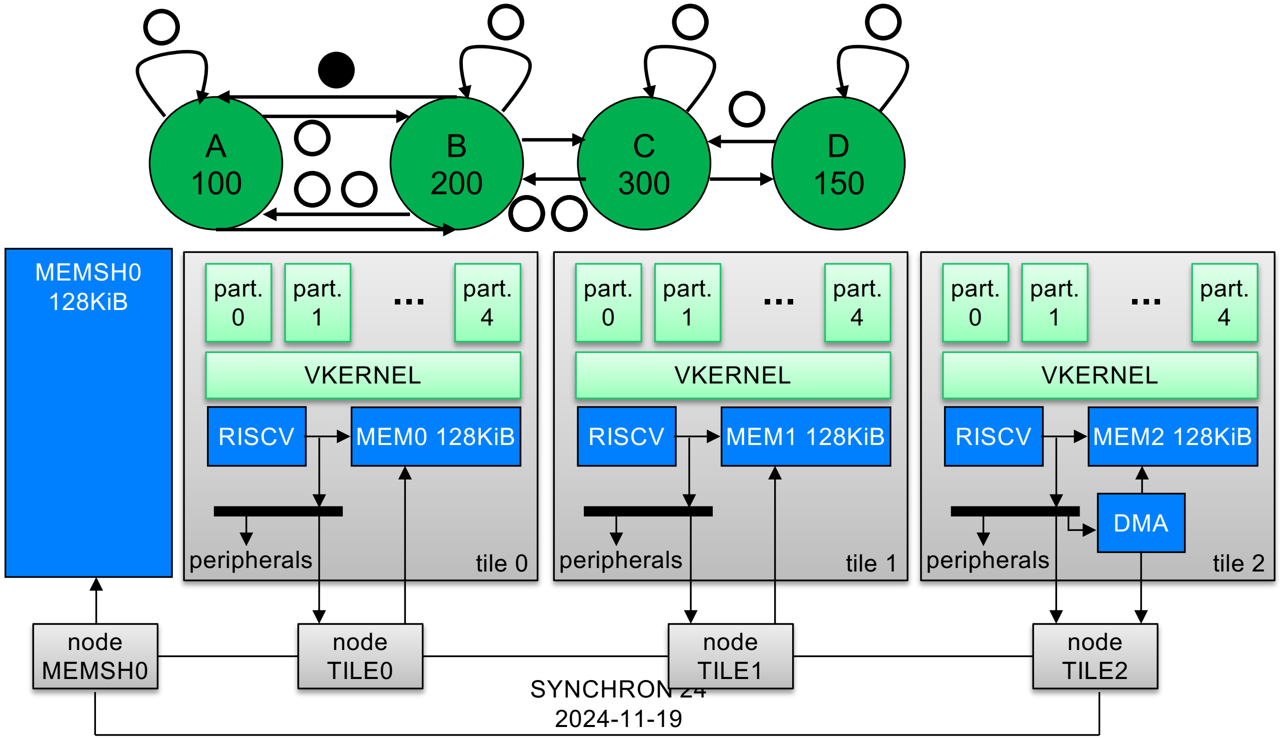
3 – mapping of actors on non-shared resources → WCET

- the microkernel virtualises actors on the processor
  - ET with microkernel is identical to their ET without microkernel (i.e. running bare metal)
- predictability and composability result in compositionality:
  - WCET of each actor is independent of & can be obtained independently of other actors



3A – mapping of actors on non-shared resources → WCET

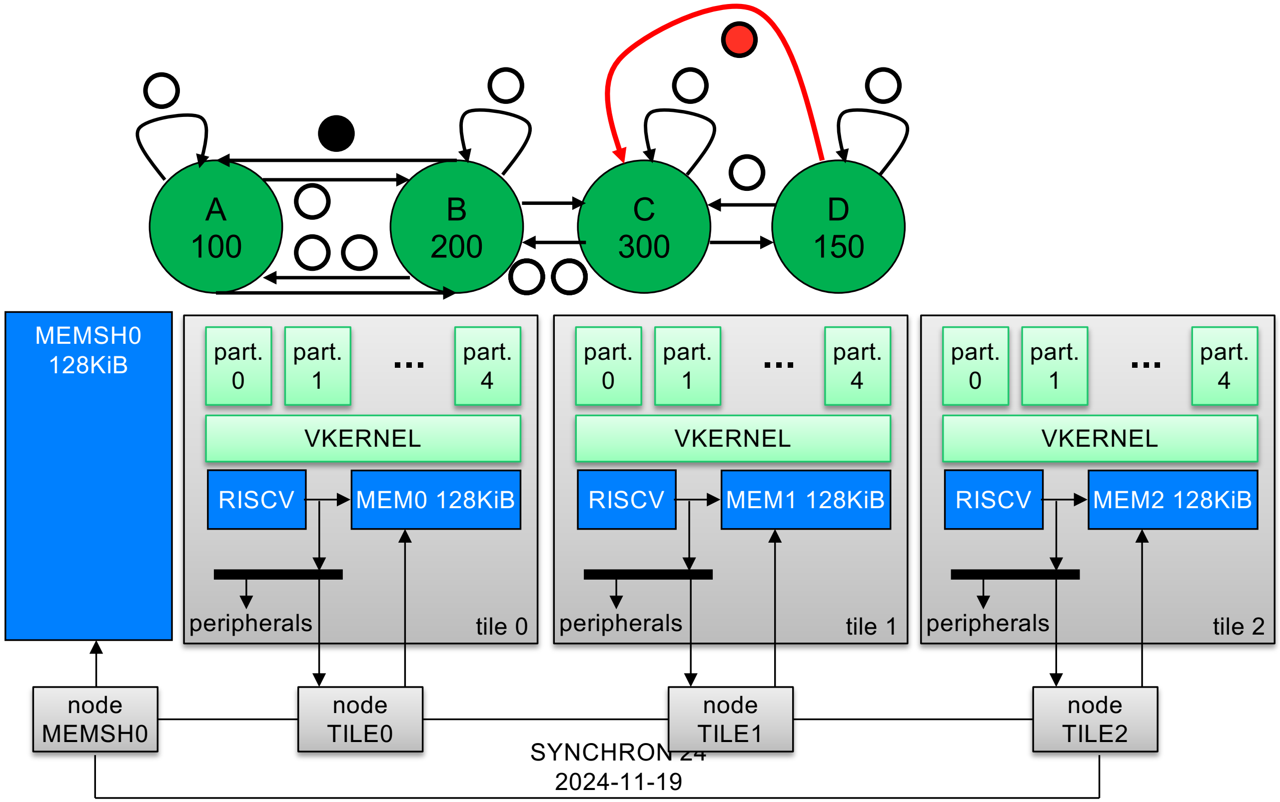
- ideally the WCET of an actor depends on a single resource only
- e.g. on a processor with loads & stores to local memory only, or use a DMA to decouple communications with remote memories



TU/e

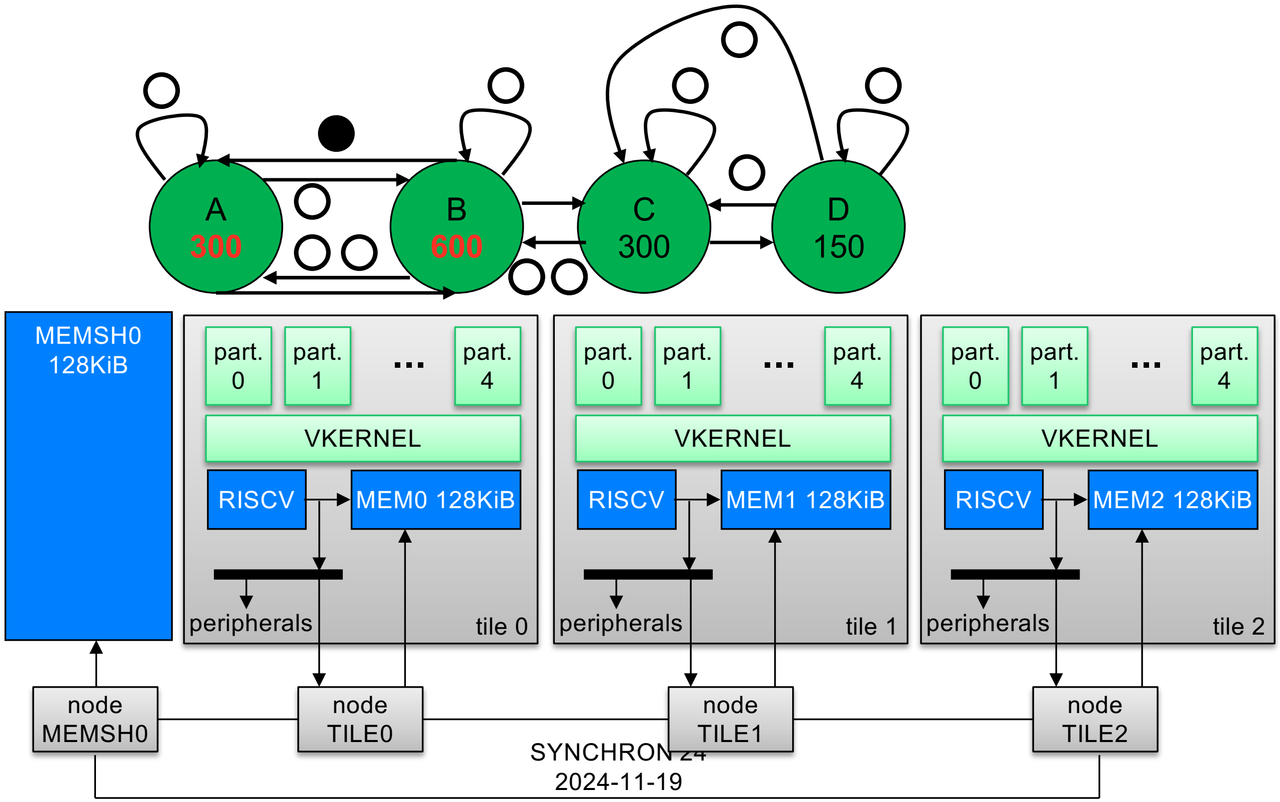
4 – adding scheduling dependencies

- suppose that actors C & D are scheduled in a static order C;D;C;D;...
  - e.g. static-order scheduler (blocking RR), or they're in the same executable
- we can model this with a scheduling dependency



5 – modelling resource sharing

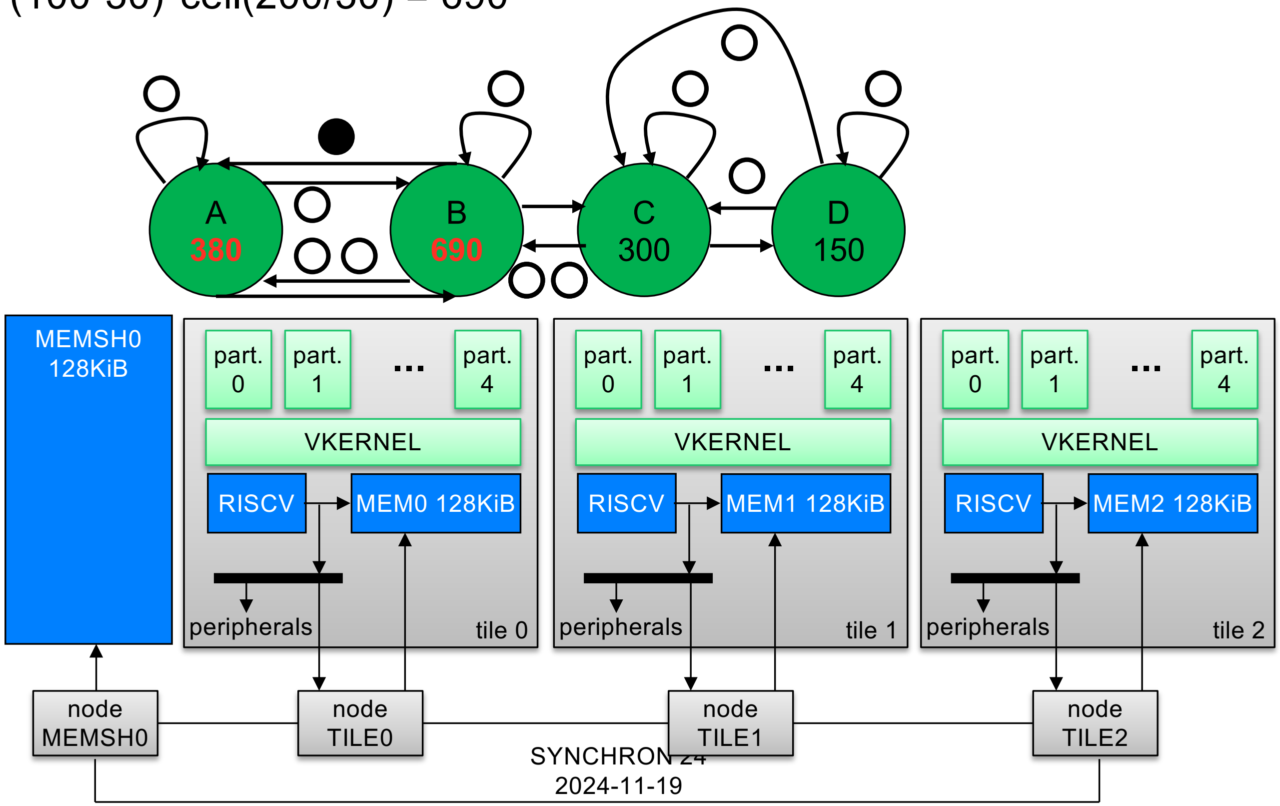
- if actors A & B run on the same resource (RISC-V) with TDM with e.g. 30% of the full capacity each
- then intuitively, they run 3x slower and their WCRT is 3*WCET
  - A 100 → 300
  - B 200 → 600



TU/e

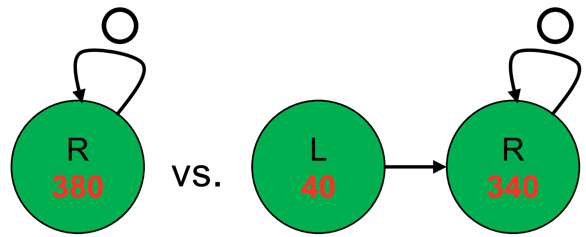
5 – modelling resource sharing

- more precisely, with TDM period P and slot size B, WCRT = WCET + (P – B) * ceil(WCET / B)
- P = 100, B = 30
- WCRT A = 100 + (100-30)*ceil(100/30) = 380
- WCRT B = 200 + (100-30)*ceil(200/30) = 690

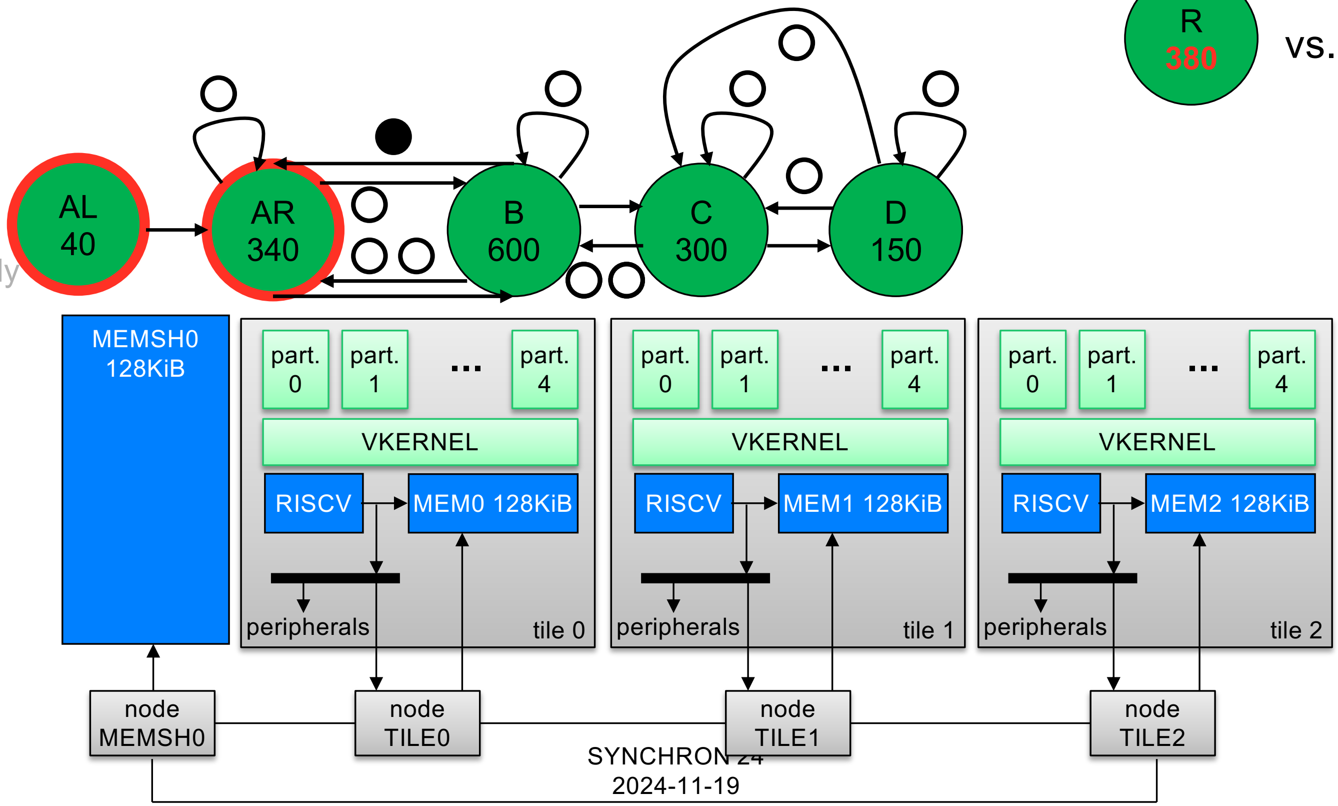


5 – modelling resource sharing

- the WCRT can model the rate of service, but then assumes a worst-case start-up latency
- we can model the arbiter as a latency-rate subgraph
- this is more accurate as we pay the initial latency only once, and get a higher rate afterwards

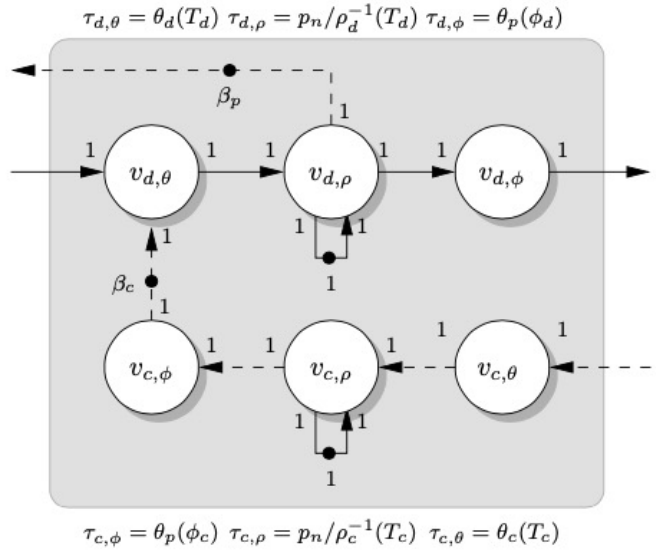


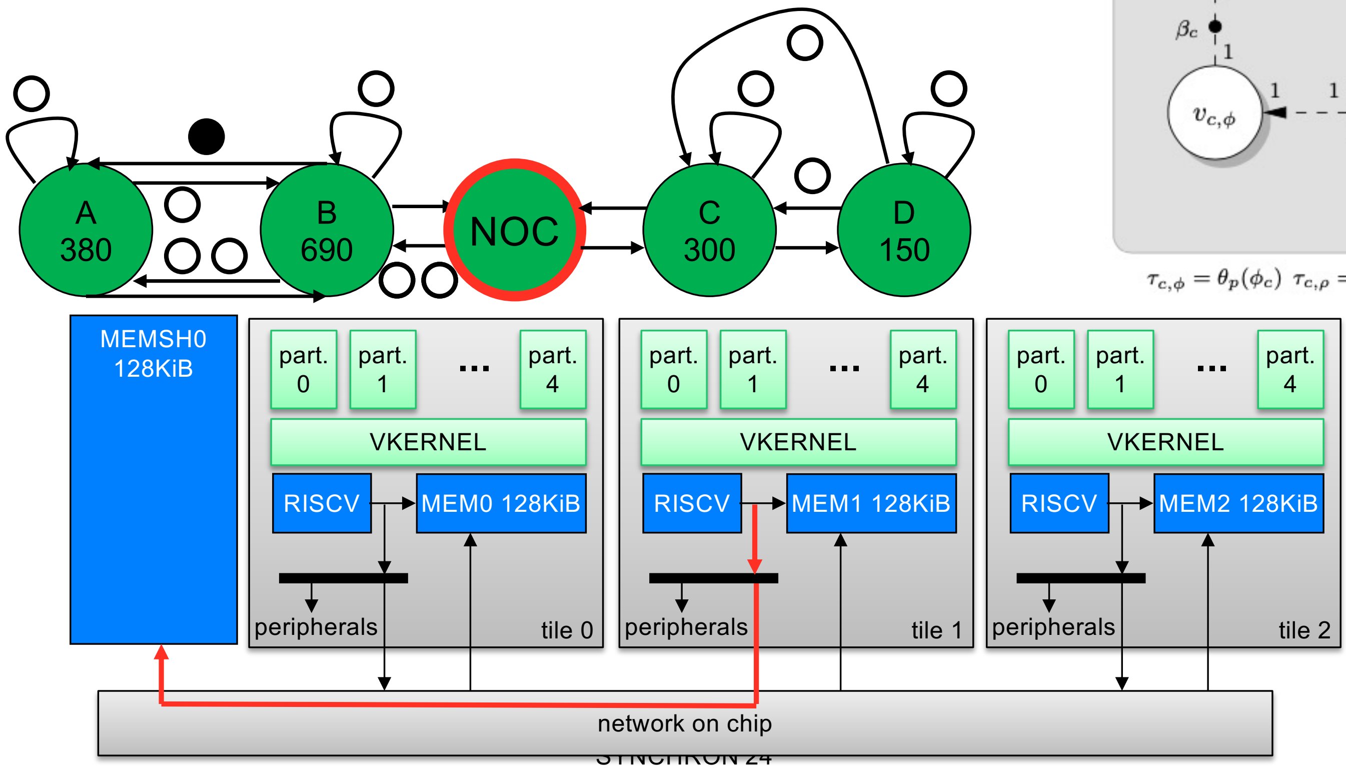
NB for illustration only – the channels between A & B are not shown correctly



5 – modelling resource sharing

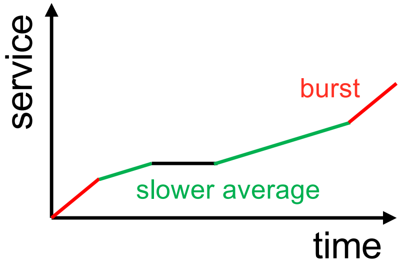
- more accurate (and complex) DF models for TDM, etc.
- similarly, a dataflow model for a network on chip connection

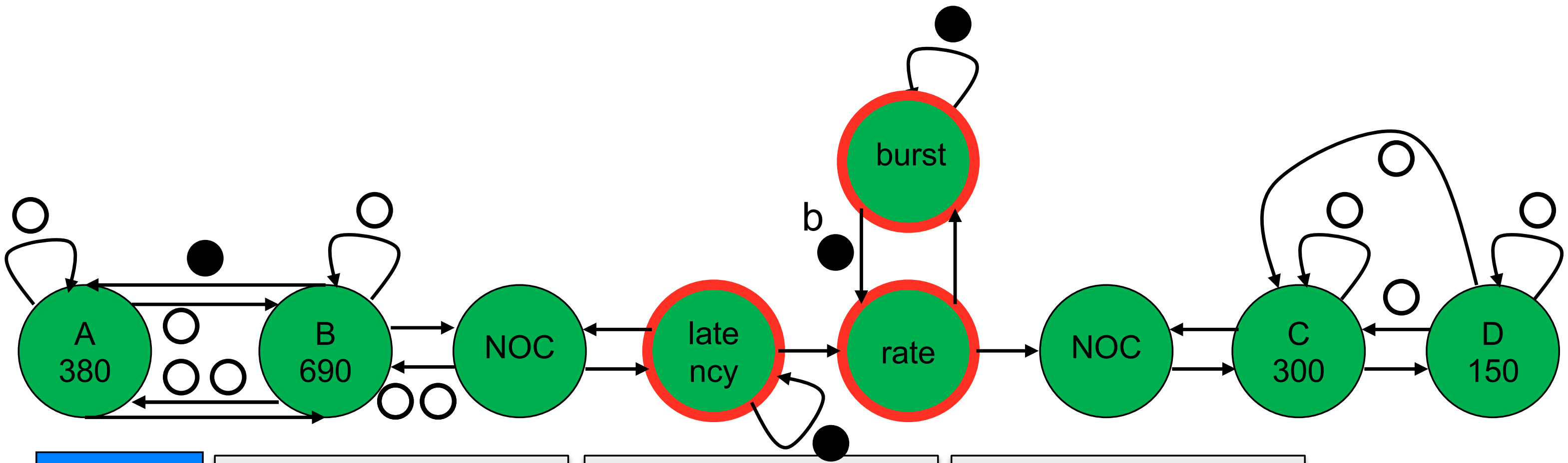


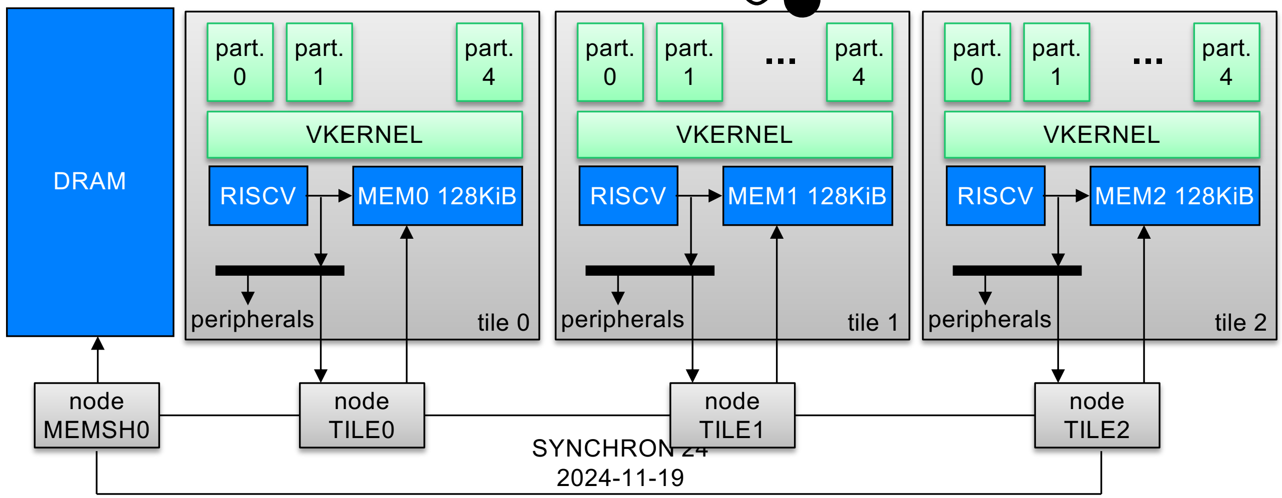


5 – modelling resource sharing

- or a DRAM arbiter with a bursty service







6 – compute the guaranteed (minimum) throughput

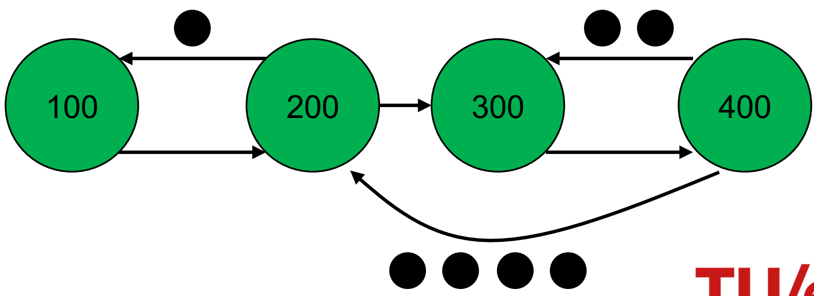
- given a dataflow graph of actors where nodes have WCRT
- the throughput (long-term average tokens per time unit) is computed by

throughput = 1 / maximum cycle mean

$$\text{MCM} = \max_{\substack{\text{c in \{all cycles} \\ \text{in the graph\}}}} \left[\frac{\text{Sum \{ WCRT of all actors in the cycle c \}}}{\text{Sum \{ of all tokens in the cycle c \}}} \right]$$

assuming self-timed execution

- intuitively, the cycle with longest WCRT limits the throughput
- tokens allow pipelining thus increasing the throughput, at the cost of more memory [NB this is monotone]



dataflow

- computation is part of the processor WCRT
- communication is part of the processor WCRT, plus optional DMA & interconnect & DRAM WCRTs
- waiting on others is taken care of by the dataflow analysis (MCM)

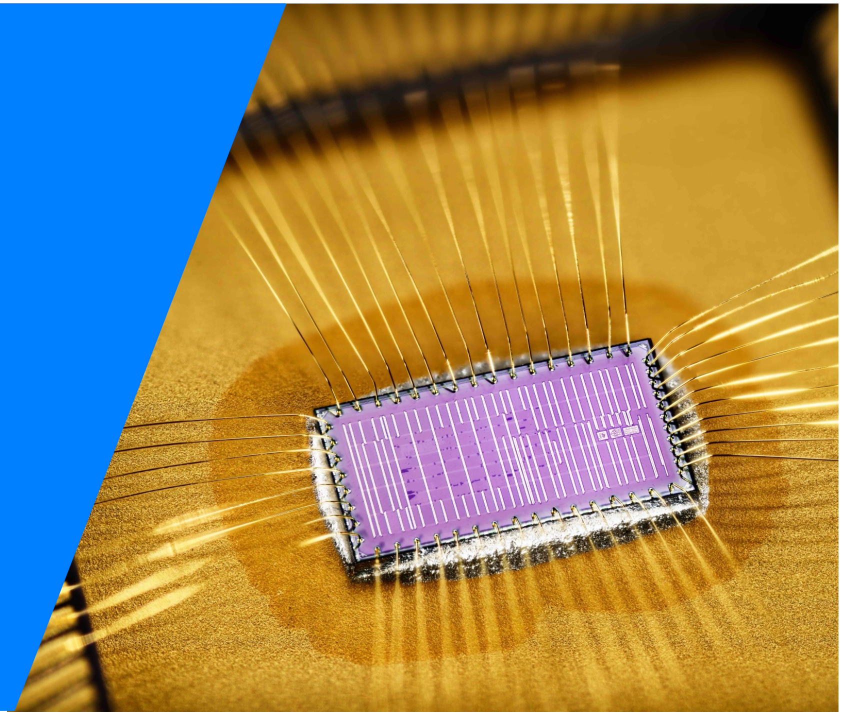
```
do {
  // read input tokens
  while (!input_token){}                                         ← waiting on others
  load input_token;                                              ← communication

  // function to compute output tokens
  (state_token, output_token) = f(state_token, input_token);     ← computation

  // write output tokens
  while (!space for output_token){}                              ← waiting on others
  store output_token;                                            ← communication
} while (true);
```

how it all fits together

- predictable resources: actor WCETs can be computed
- predictable arbitration: aka budget schedulers: actor WCRT can be computed from WCET
- compositionality: WCET & WCRT of actors are independent, but their ETs need not be
  → verify worst-case performance piece-wise, rather than monolithically when all applications are available
- composability: application ETs are independent
  → verify actual-case performance independently per application, e.g. for adaptive applications
- determinism: with the same input, application execution trace and ET remain the same in every run, whether running bare metal or in a VEP

- all the above hold for a mix of real-time & non real-time applications
- for debug, composability helps, and determinism helps even more

Design flow

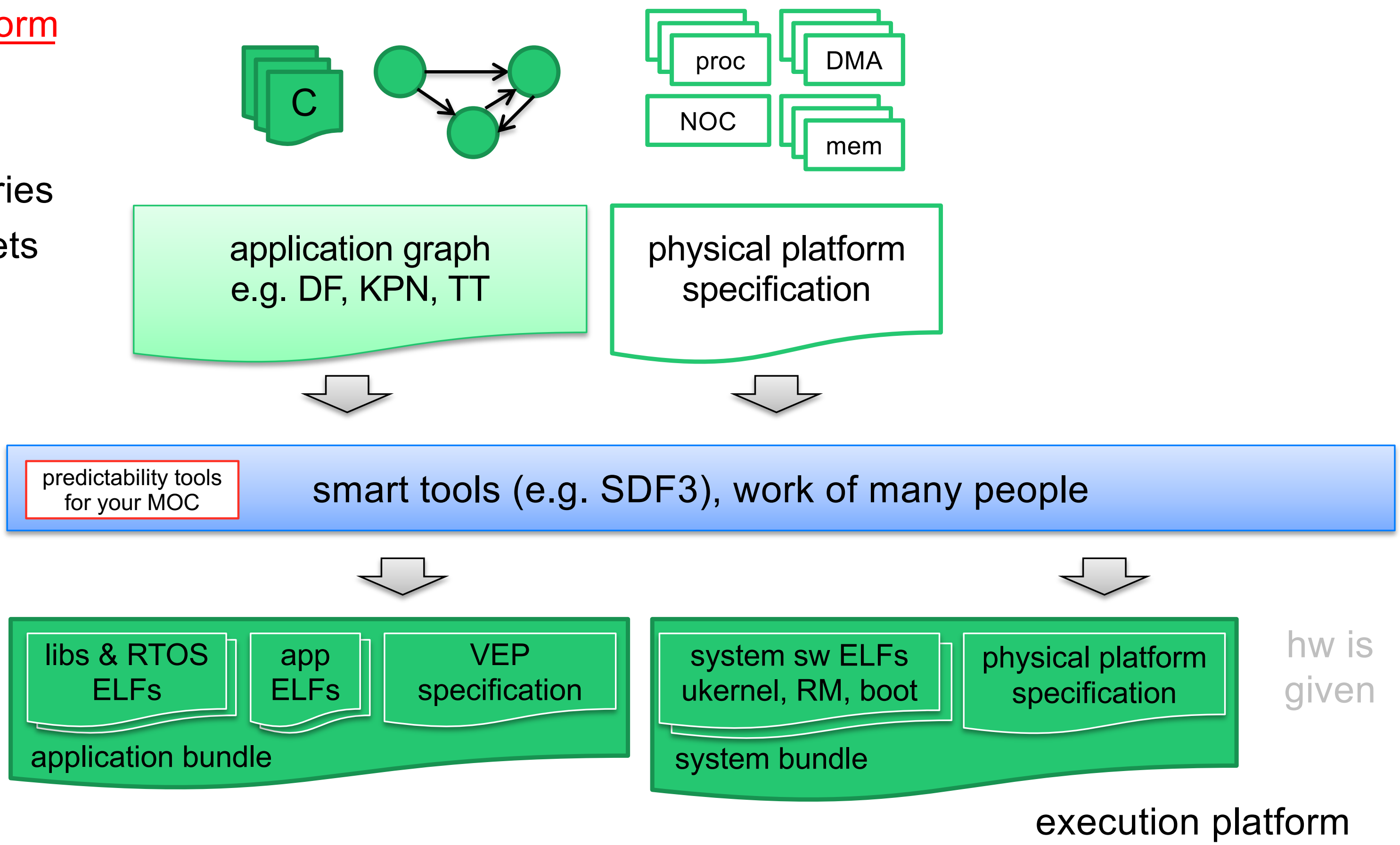


Kees Goossens <k.g.w.goossens@tue.nl>
Electronic Systems Group
Electrical Engineering Faculty

EINDHOVEN UNIVERSITY OF TECHNOLOGY

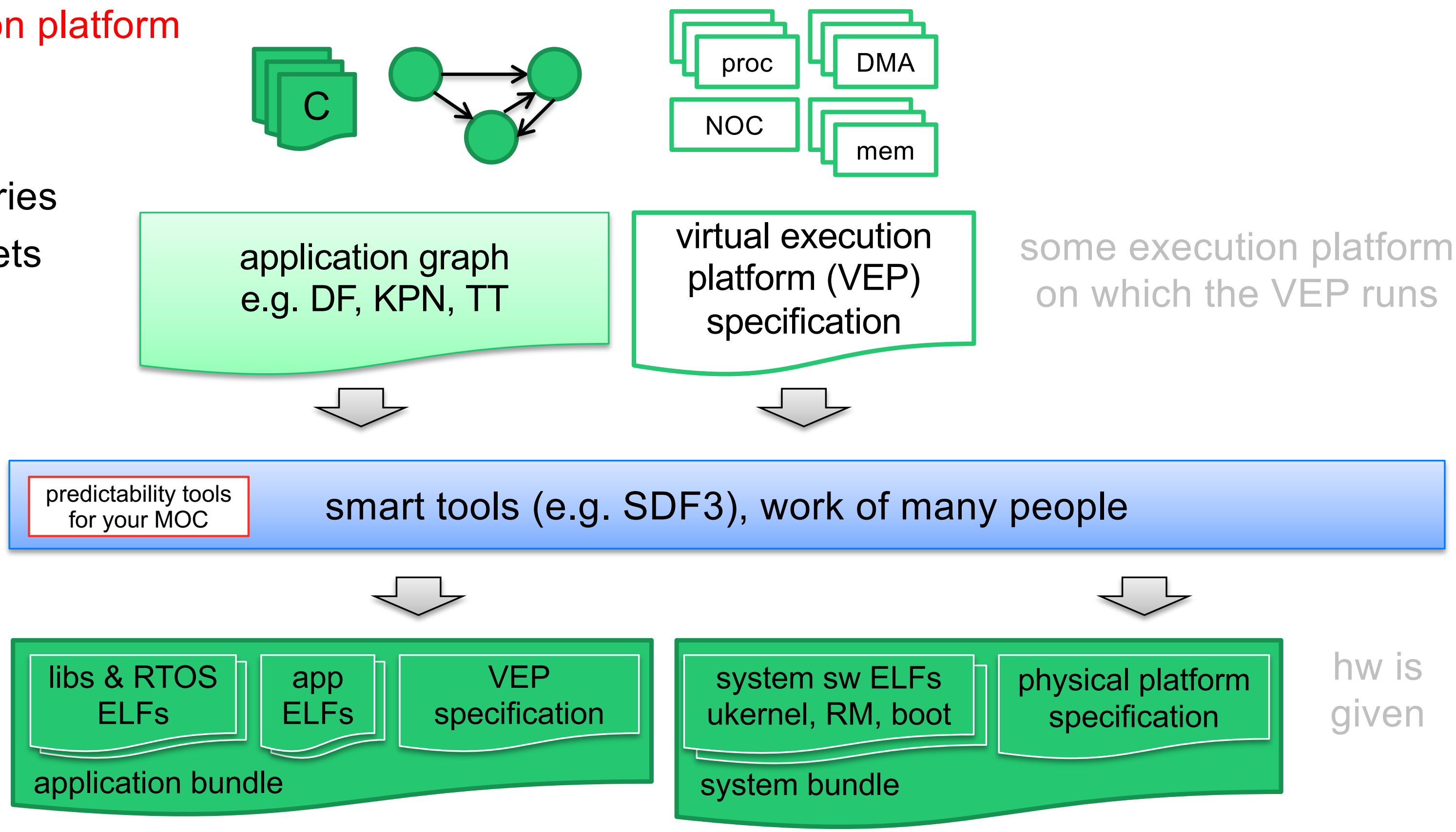
CompSOC design flow (mapping) – in the past, not operational now

- map to given execution platform
- produces a "bundle"
  - application + system binaries
  - VEP specification = budgets
    - mapping
    - MMU settings
    - arbiter settings
    - NOC paths, etc.
- was operational for SDF



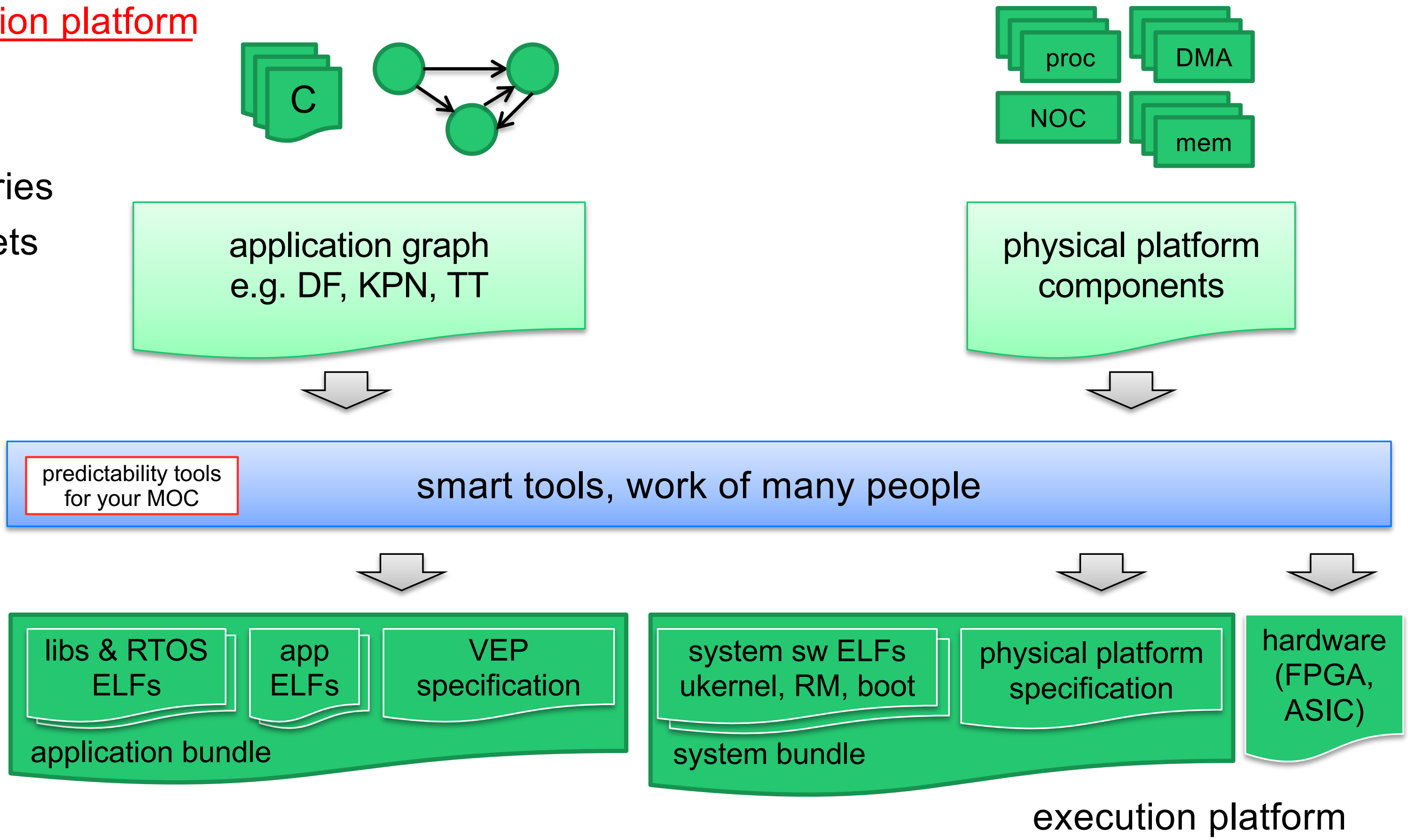
CompSOC design flow (mapping) – in the past, not operational now

- map to given virtual execution platform
- produces a “bundle”
  - application + system binaries
  - VEP specification = budgets
    - mapping
    - MMU settings
    - arbiter settings
    - NOC paths, etc.
- was operational for SDF

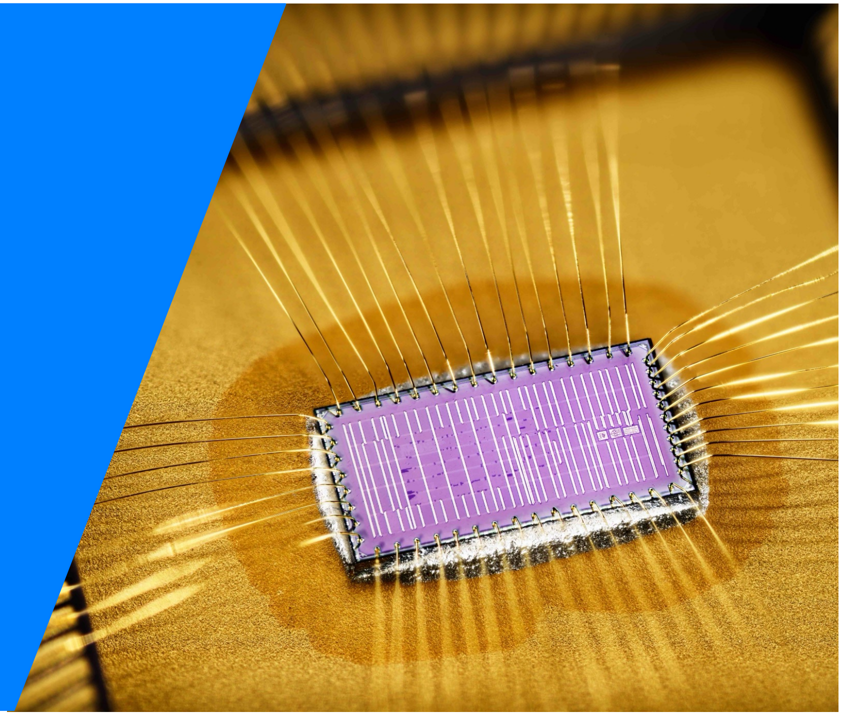


CompSOC design flow (synthesis) – in the past, not operational now

- synthesise a virtual + execution platform
- produces a "bundle"
  - application + system binaries
  - VEP specification = budgets
    - mapping
    - MMU settings
    - arbiter settings
    - NOC paths, etc.
- produces platform hardware
  - e.g. used to generate the ASIC hardware



Some examples



Kees Goossens <k.g.w.goossens@tue.nl>
Electronic Systems Group
Electrical Engineering Faculty

wait-free FIFO [e.g. Implementing Dataflow With Threads, Lamport, 2005]


```
void fifo_write_token(fifo_t volatile * const fifo_admin, void const * const token_to_write)
{
  void volatile * t;
  do {
    t = (fifo_admin->tokens + fifo_admin->wi * fifo_admin->token_size);
  } while (t == NULL)
  memcpy(t, token_to_write, fifo_admin->token_size);
  fifo_admin->wc++;
  fifo_admin->wi = (fifo_admin->wi+1) % fifo_admin->nr_tokens;
}
```

wait on others

communication,
part of actor WCET

- predictable → used in e.g. SDF

- barrier, publish-subscribe are similar
- for wait-free buffer (sample & hold) use Simpson's algorithm
- for mutexes use Peterson's, Lamport's bakery algorithms

wait-free FIFO [e.g. Implementing Dataflow With Threads, Lamport, 2005]


```
void fifo_write_token(fifo_t volatile * const fifo_admin, void const * const token_to_write)
{
  void volatile * t;
  do {
    t = (fifo_admin->tokens + fifo_admin->wi * fifo_admin->token_size);
  } while (t == NULL)
  memcpy(t, token_to_write, fifo_admin->token_size);
  fifo_admin->wc++;
  fifo_admin->wi = (fifo_admin->wi+1) % fifo_admin->nr_tokens;
}
```

wait on others

communication,
part of actor WCET

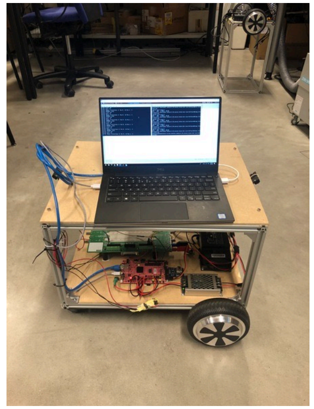
- predictable → used in e.g. SDF

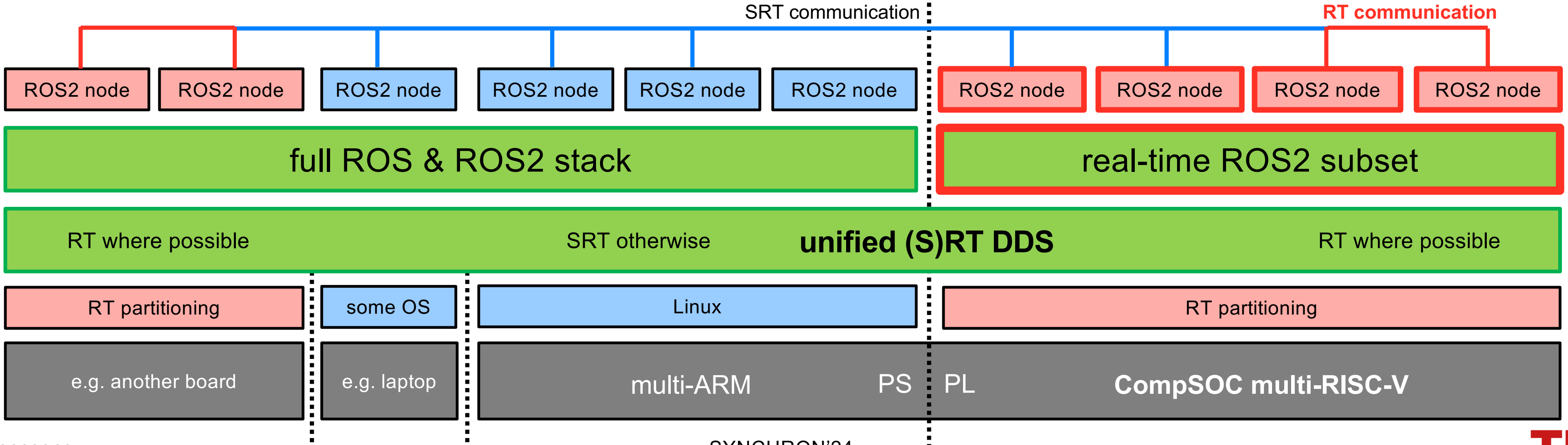
- CompSOC has no fences, atomic instructions, etc. but is sequentially consistent
- C compilers implement sequential consistency on atomic variables only
- but by using C volatile (to disallow reordering) with CompSOC's sequential consistency we can write thread-safe mutexes, locks, FIFOs, etc.

distributed CompSOC using ROS2

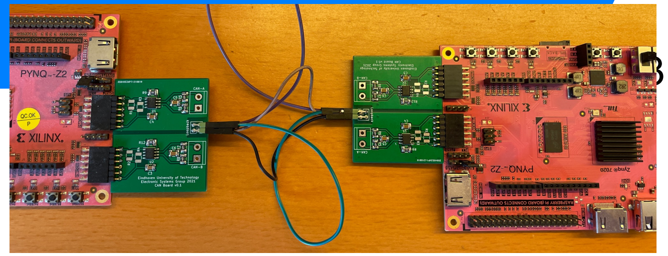
- Data Distribution Service (DDS)
- Real-Time Publish Subscribe (RTPS)
- Extremely Resource Constrained Environment (XRCE)

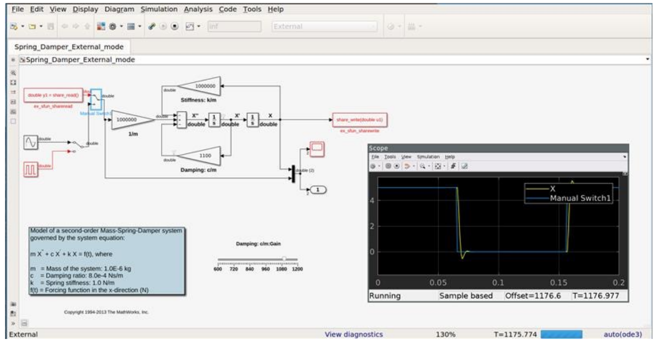
- real-time wait-free pub-sub on CompSOC
- non-real-time pub-sub for CompSOC ←→ Linux





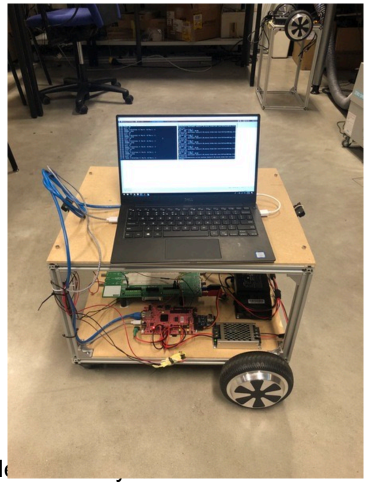
use of CompSOC in TUE curriculum

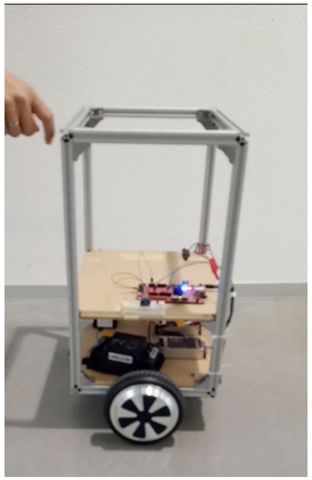


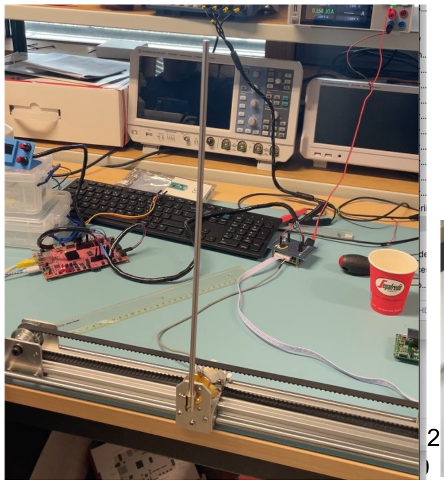


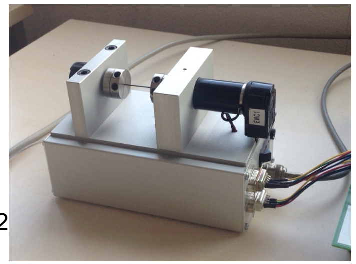
- programming CAN
- embedded control (with Simulink PIL, HIL)
- real-time multicore programming

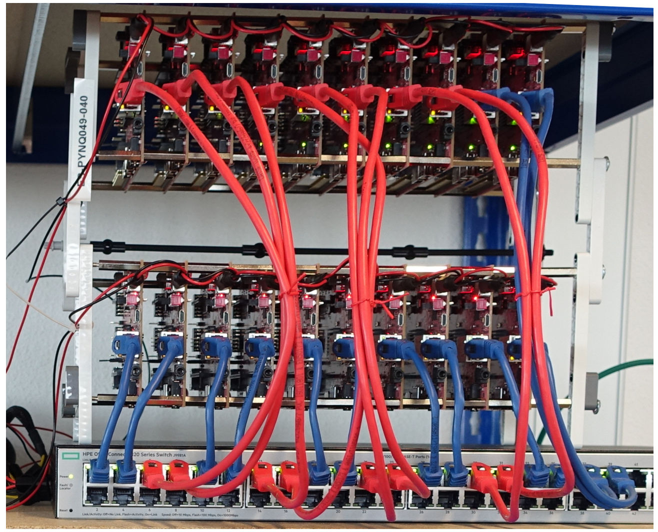
- embedded-control research





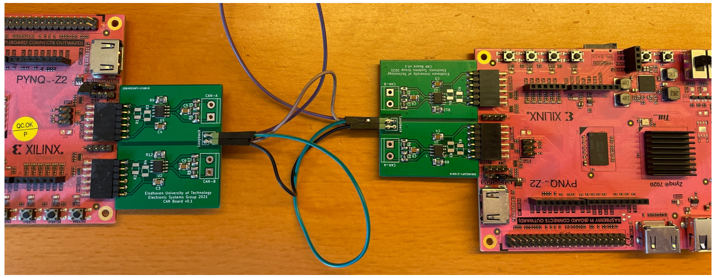


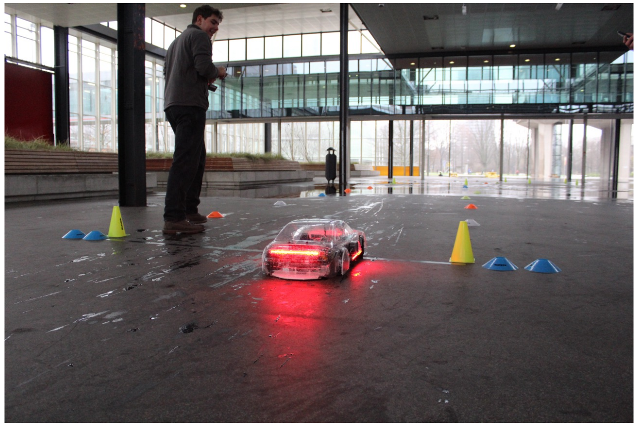




use of PYNQ throughout TUE EE curriculum (pynq.tue.nl)

- C programming course on Linux on Arm
- Verilog design on FPGA + Linux drivers on Arm
- programming cradles, cars, autonomous robots
- programming AUTOSAR





questions?

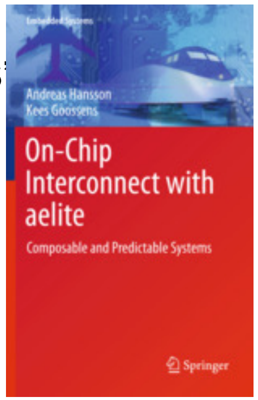
- demos
- availability of CompSOC/Verintec platform
- ...

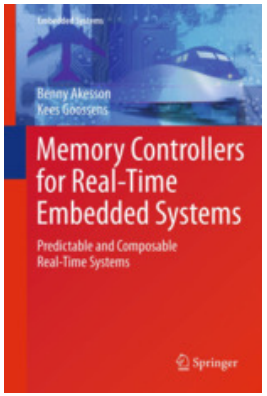
more information: http://www.es.ele.tue.nl/~kgoossens/research.html

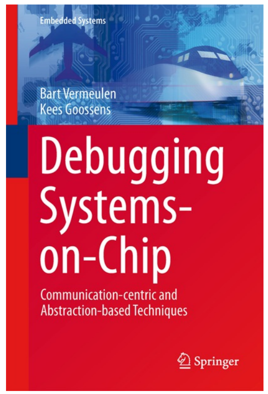
- CompSOC overview *NoC-Based Multiprocessor Architecture for Mixed-Time-Criticality Applications*, in Springer Handbook of Hardware/Software Codesign, 2017

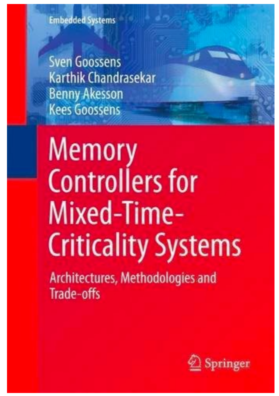
- A Deployment Framework for Quality-Sensitive Applications in Resource-Constrained Dynamic Environments, DSD’21
- CompROS: A composable ROS2 based architecture for real-time embedded robotic development, IROS’21
- Resource Utilization and Quality-of-Control Trade-Off for a Composable Platform, DATE’16
- State-based switching multi-rate controller for improving resource utilization on predictable and composable platforms, MICPRO’22

- Chip health monitoring with in-situ delay monitoring, DATE’19
- Aethereal real-time NOC, DAC’10
- Real-time DRAM memory controller, DATE’13, ECRTS’
- CompOSe RTOS, MICPRO’11
- CoMiK microkernel, DATE’14
- Composable power management, SAMOS’11
- Design flow, FPGA World’13
- SDF3, DAC’06
  http://www.es.ele.tue.nl/sdf3/

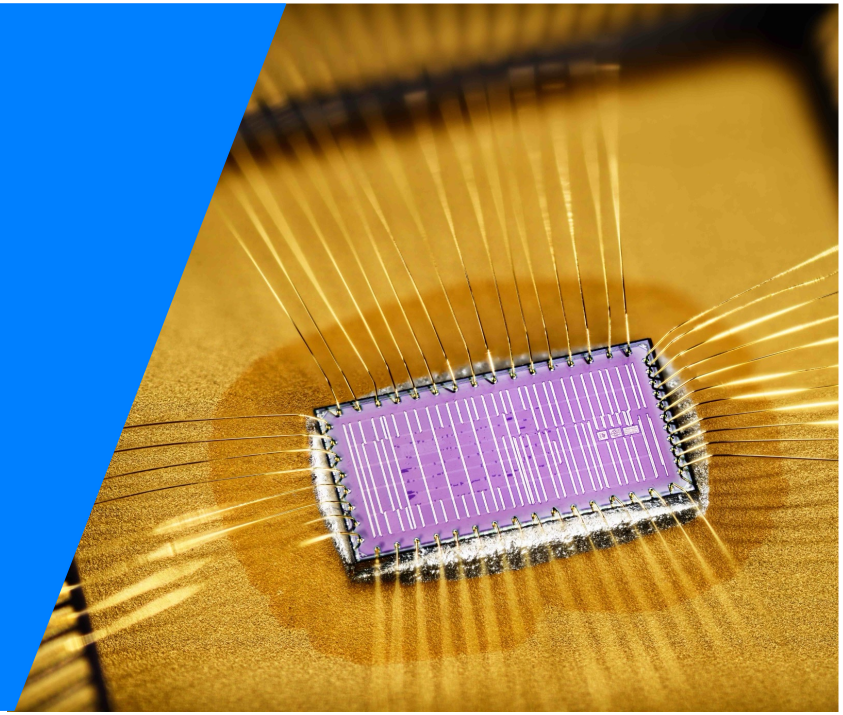








Resource-Aware component model



Kees Goossens <k.g.w.goossens@tue.nl>
Electronic Systems Group
Electrical Engineering Faculty

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Logical Methods in Computer Science
Volume 17, Issue 2, 2021, pp. 19:1–19:34
https://lmcs.episciences.org/

Submitted Feb. 20, 2020
Published May 26, 2021

INTERFACE MODELING FOR QUALITY AND RESOURCE MANAGEMENT *

MARTIJN HENDRIKS [a,b], MARC GEILEN [a], KEES GOOSSENS [a], ROB DE JONG [c], AND TWAN BASTEN [a,b]

[a] Eindhoven University of Technology, Eindhoven, The Netherlands

[b] ESI (TNO), Eindhoven, The Netherlands

[c] Philips Medical Systems International BV, Best, The Netherlands

Abstract. We develop an interface-modeling framework for quality and resource management that captures configurable working points of hardware and software components in terms of functionality, resource usage and provision, and quality indicators such as performance and energy consumption. We base these aspects on partially-ordered sets to capture quality levels, budget sizes, and functional compatibility. This makes the framework widely applicable and domain independent (although we aim for embedded and cyber-physical systems). The framework paves the way for dynamic (re-)configuration and multi-objective optimization of component-based systems for quality- and resource-management purposes.

formal resource management: budgets

- to be predictable, resource usage must be budgeted & enforced
- composable budgets must be non work conserving (think: TDM)
- best-effort applications also have budgets, but with less good service

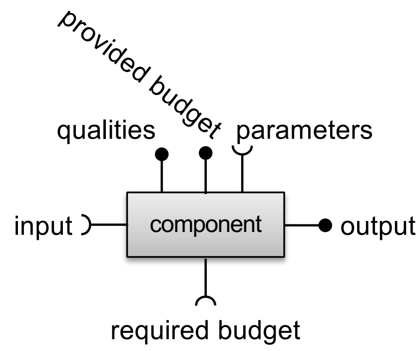
- a budget is binary: you either get it or you don't
- when you get it, it is guaranteed until you relinquish it

- reserving a budget on a resource results in a virtual resource
- budgets & (virtual) resources are hierarchical
  - platform > tile > processor > DMEM

- a virtual execution platform is a set of virtual resources

formal resource management: components

- component
  - set of configurations
    - parameter (determines the configuration)
    - quality (cost, performance)
    - output
    - input
    - provided budget
    - required budget
    - initial state (required to instantiate component)



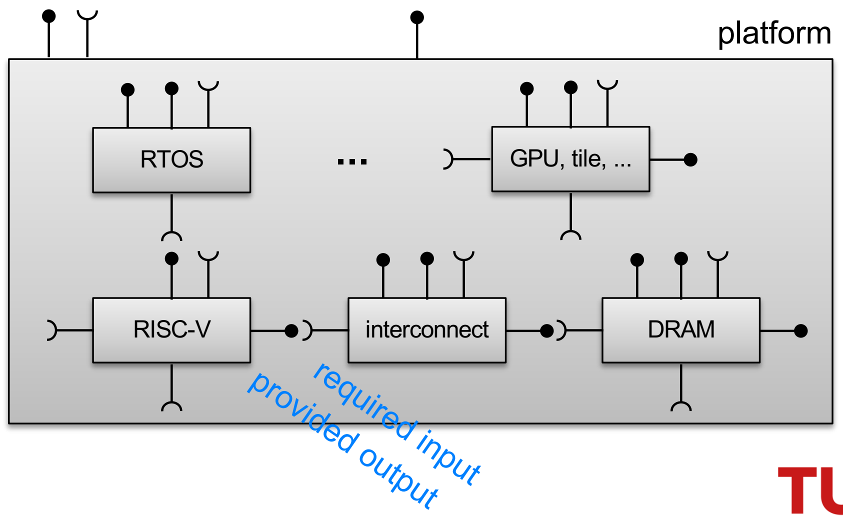
- configurations capture
  - different modes (e.g. single video, picture in picture; sleep vs. burst mode)
  - different implementations & different mappings (at different cost:performance points)
  - Pareto set

component composition

- components can be composed
  - horizontally: I/O
  - vertically: provided – required budgets (services)
  - hierarchically

- application
- virtual execution platform
- execution platform

(hierarchical) set of components

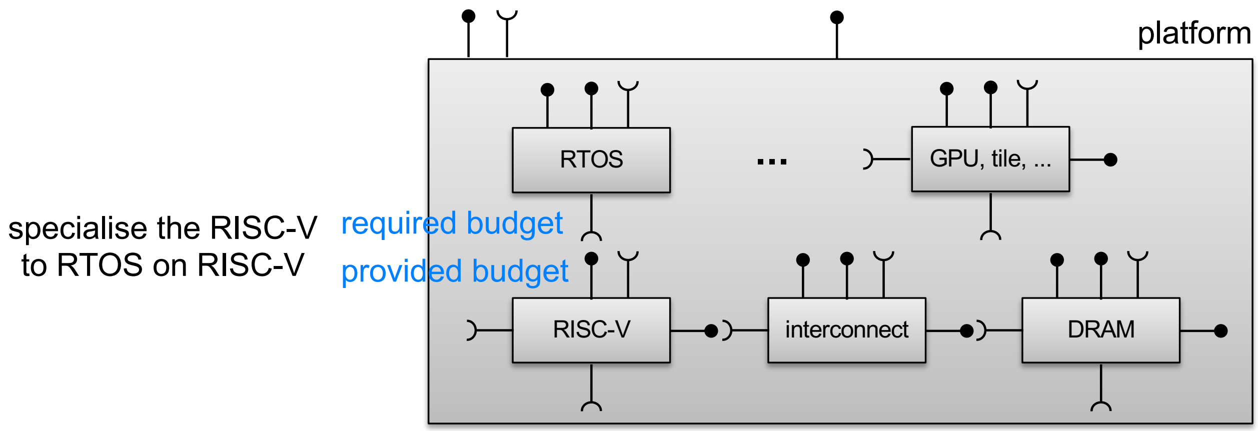


component composition

- components can be composed
  - horizontally: I/O
  - vertically: provided – required budgets (services)
  - hierarchically

- application
- virtual execution platform
- execution platform

(hierarchical) set of components



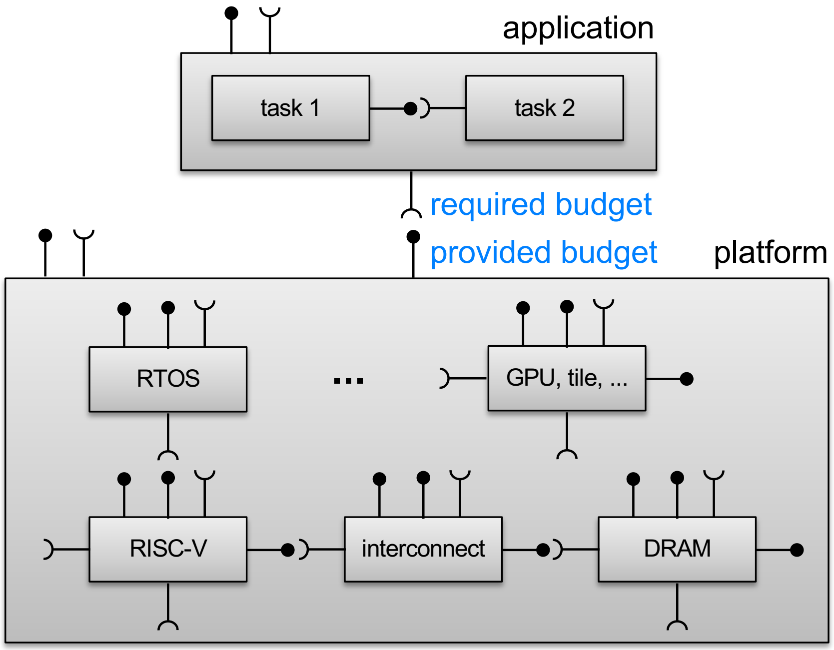
component composition

- components can be composed
  - horizontally: I/O
  - vertically: provided – required budgets (services)
  - hierarchically

- application
- virtual execution platform
- execution platform

} (hierarchical) set of components

running a single application on a platform

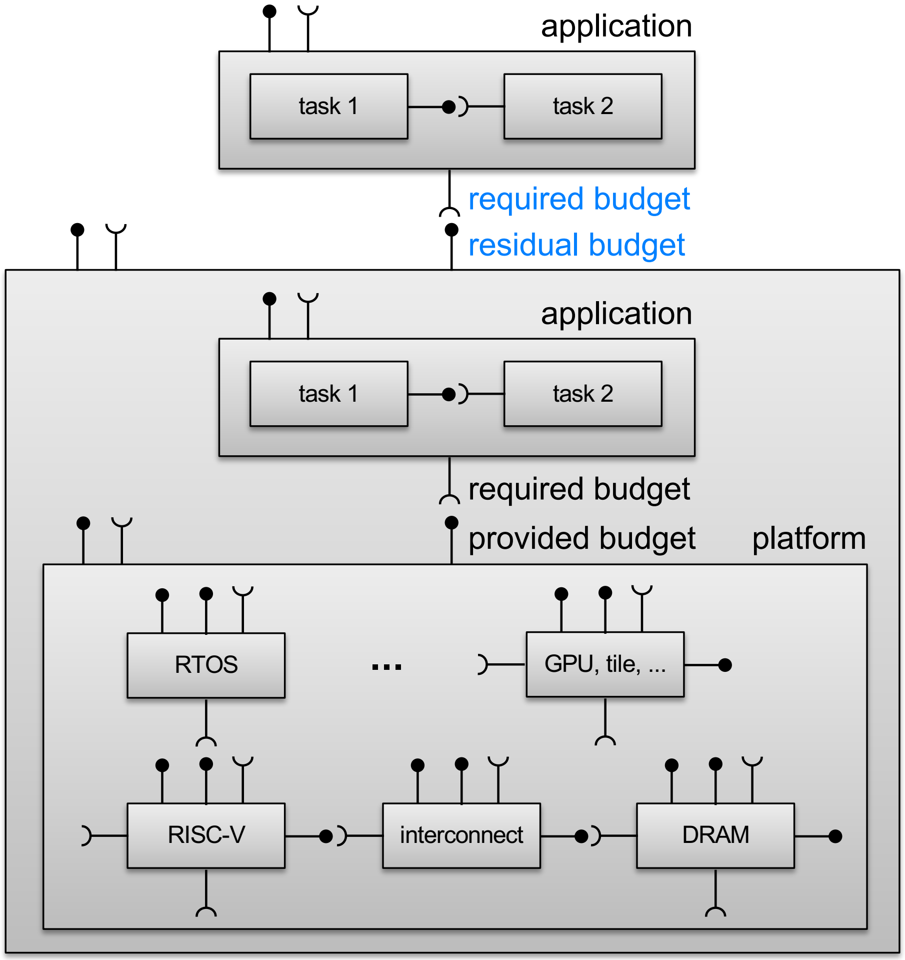


component composition

- components can be composed
  - horizontally: I/O
  - vertically: provided – required budgets (services)
  - hierarchically

- application
- virtual execution platform
- execution platform

} (hierarchical) set of components

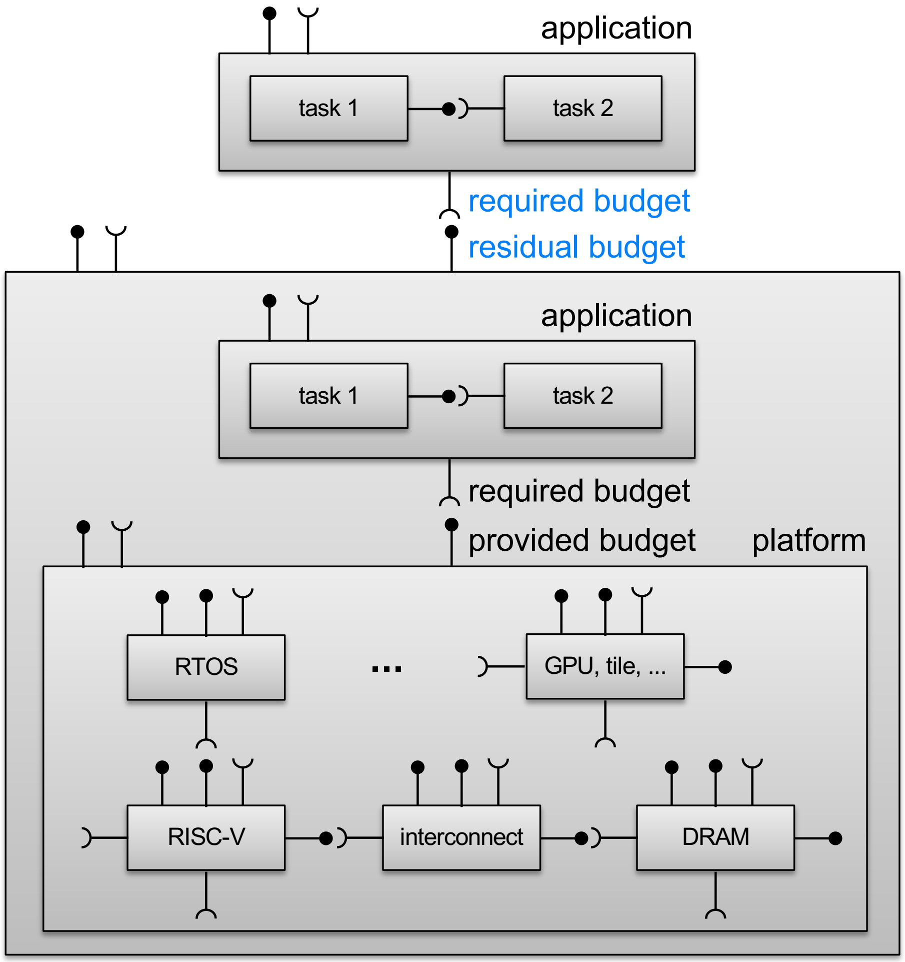


component composition & optimisation

- composition requires

1. budget matching
   - output >= input
   - provided >= required

2. satisfying user parameter & quality constraints
   - e.g. high resolution, max. 10 Watt



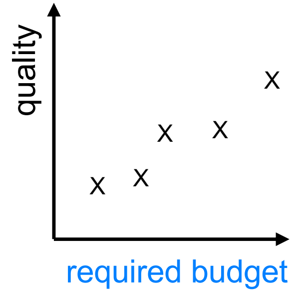
component composition & optimisation with Pareto algebra

- composition requires

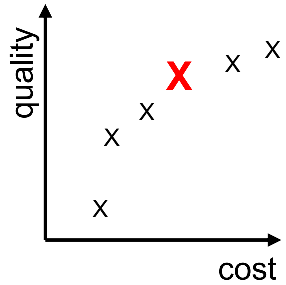
1. budget matching
   - output >= input
   - provided >= required
2. satisfying user parameter
   - e.g. high resolution, m
3. Pareto optimisation
   - pick best configuration with high resolution, max. 10 Watt
   - use Z3 SMT solver at design time (or in cloud @ run time) or run-time embedded heuristics

   → declarative description of the best VEP(s)
4. deploy the VEP

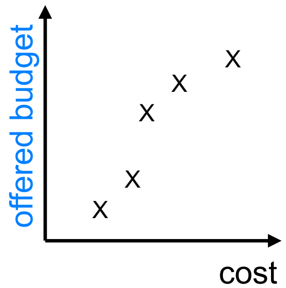

```
"components":
[
  {
    "id" : "Task1",
    "configurations":
    [{
      "inputs":[{"raw_frames" : "30Hz"}, ...],
      "outputs":["processed_frames" : "30Hz"}, ...],
      "parameters":[{"resolution" : "720p"}, ...],
      "qualities":[{"framerate" : 30}, ...],
      "required_budget”:
       {
        "TILE": { "RISCV":
                  [{
                    "unit" : "cycles",
                    "type" : "average_rate",
                    "value" : 100K
                   },... // other services from RISCV
                  ], ... // other resources from TILE
                 }, ... // other resources besides TILE
       },
      "initial_state":[{"IDMEM" : ".../task1.hex"}, ...]
    }, ... // other application configurations
   ]}, ... // other components
],
"compositions":
[
  "App1 = Task1 => Task2",
   ...
]
```

application



optimise
(application X execution platform)



execution
platform

haskell demo

- some values
- tta
- osC, app1C, vectorN
- compsoc
- ttcan
- sdn
- abstraction