Predictability in the CoMPSoC platform - processor-tile

Anca Molnos[1], Andrew Nelson[1], Ashkan Beyranvand Nejad[1], Sander Stuijk[2], Martijn Koedam[2], Kees Goossens[2]

[1] Delft University of Technology, the Netherlands

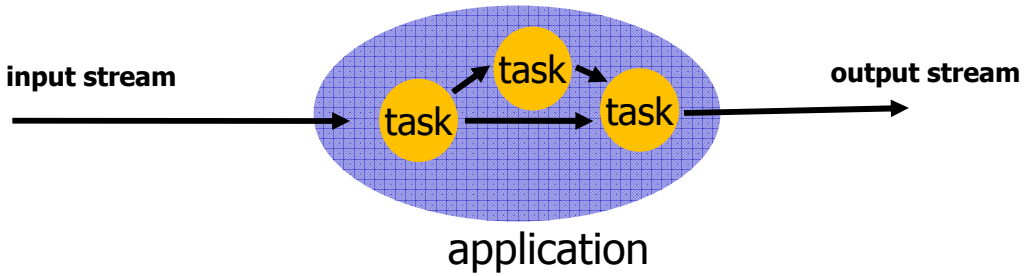
[2] Eindhoven University of Technology, the Netherlands

Overview

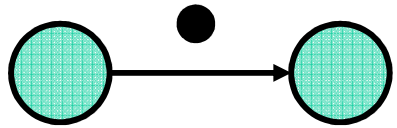
- application model
- predictable processor-tile architecture
- processor sharing: compOSe
  - scheduling
  - APIs
- conclusions

Recap: context

- FRT, SRT, NRT applications running concurrently on an MPSoC
- FRT application model: streaming
  - tasks that communication through blocking FIFOs/circular buffers.
  - demand formal performance analysis (latency, throughput guarantees)



Dataflow models



- fit streaming applications
- nodes: actors
- edges: unbounded queues between actors
- dots: tokens
- actors have firing rules
- execution time from firing to completion
  - no blocking during execution

- any graph, cycles are allowed
- single rate, multi-rate, cyclo-static (CSDF)
- dynamic / variable-rate dataflow

Tasks and firing

Just an implementation

```
while(1) {
  read FIFO1
  compute1(…)
  read FIFO2
  write FIFO3
  compute2(…)
  …
  write FIFO4
  …
}
```

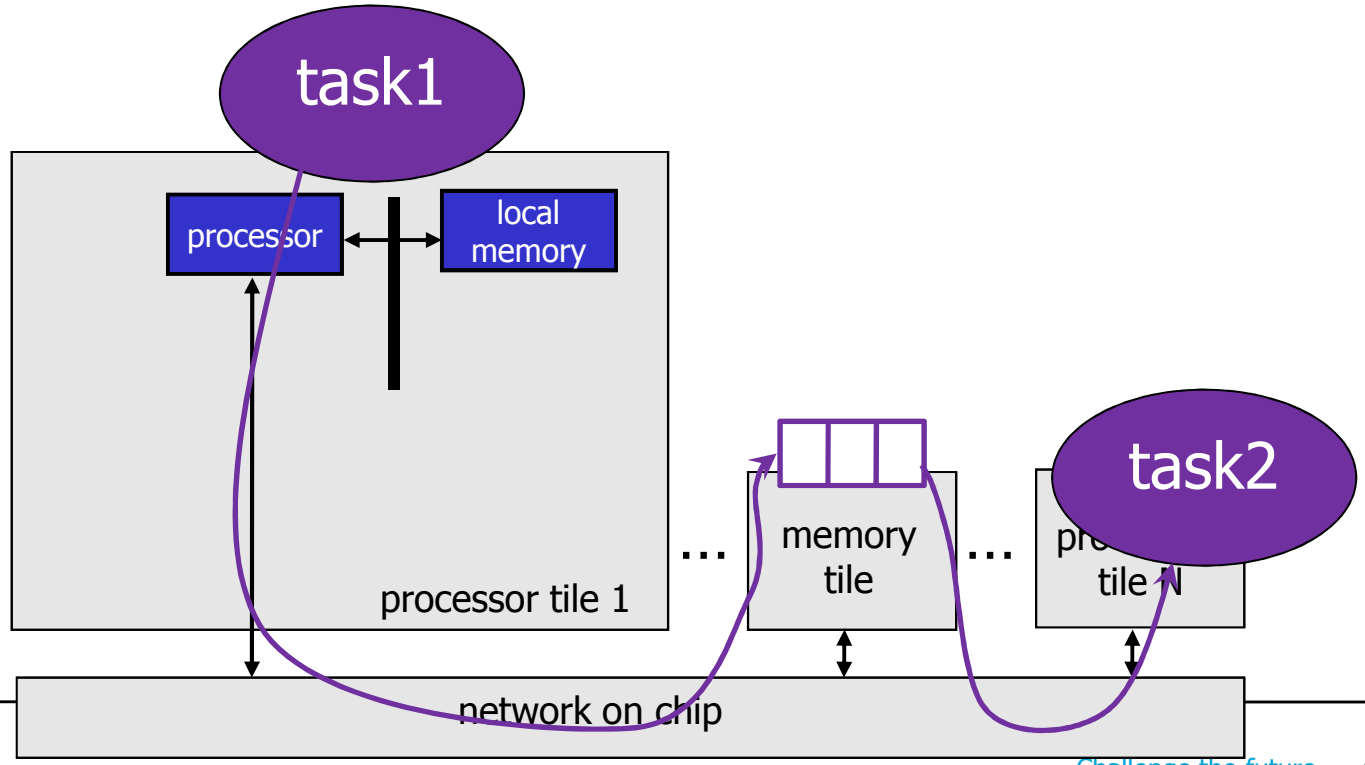
Dataflow-friendly implementation

```
while(1) {
  //firing rules check
  if (data&space) {
   read FIFO1
   read FIFO2
   …
   //actual 'task'
   compute(…)
   …
   write FIFO3
   write FIFO4 }}
```

- the code within the if statement could execute without blocking the processor, following the dataflow model

Application on the architecture

- `compute` – on the processor, local memory, and potentially also NoC, remote memory.
- `read/write FIFO` – local memory, NoC, remote memory



Performance analysis (requirements)

To analyze such an application end-to-end we need to bound the time spend in `read,compute,write:`

1. predictable resources: bounds on execution time
2. predictable sharing: bounds on response time
   - predictable arbiter
   - predictable resource state between requestors.

Requests executed at resources

- task (compute) → processor
  - task = set of instructions
  - some instructions: load&store may result in transactions (NoC, memory).
  - WCET analysis should work
- transaction → NoC
  - predictable: guaranteed maximum latency, minimum throughput
- transaction → memory
  - predictable: guaranteed maximum latency, minimum throughput

Ideally:

- should not model each instruction entire system analysis.
- tight bounds (accurate models)

Performance analysis (extra requirement)

3. no inter-resource dependencies (decouple resources and their analysis models).
   - `compute` does not use multiple resources.
     - WCET analysis
   - `read` may result in NoC & memory requests
     - the processor has to wait for these data
   - `write` may result in NoC & memory requests
     - the processor should not block for these requests (posted writes), hence the communication should be performed in a separate thread (also a composability request)

Processor-tile design choices

Processor architecture:

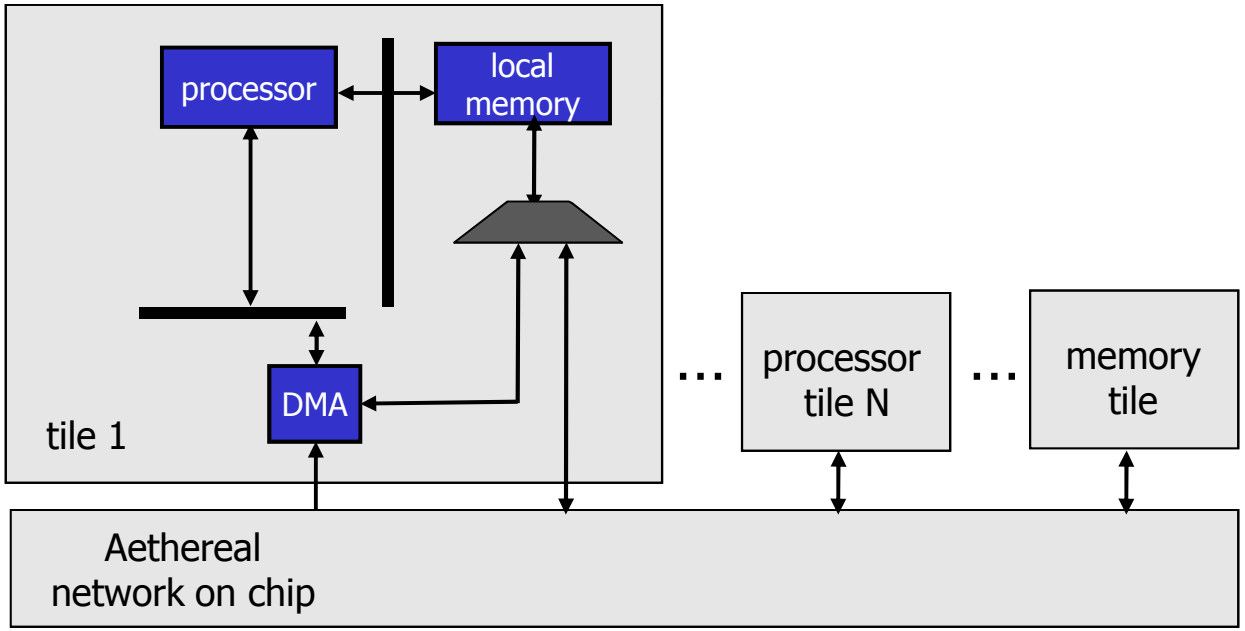
1. discard features that are not predictable, e.g., OoO, caches with random replacement, etc. (to bound the `compute` time)

Memory hierarchy:

2. task code and data fit in local memory
   - `compute` utilizes only the processor
   - optionally: tasks pre-fetched in and swapped out tile
3. inter-tile communication via DMAs
   - no DMA interrupts: the processor polls for DMA ready
   - `reads` are interruptible (for composability)
   - DMA with "deep" request queues, so that tasks don't block for `writes` (optimization)

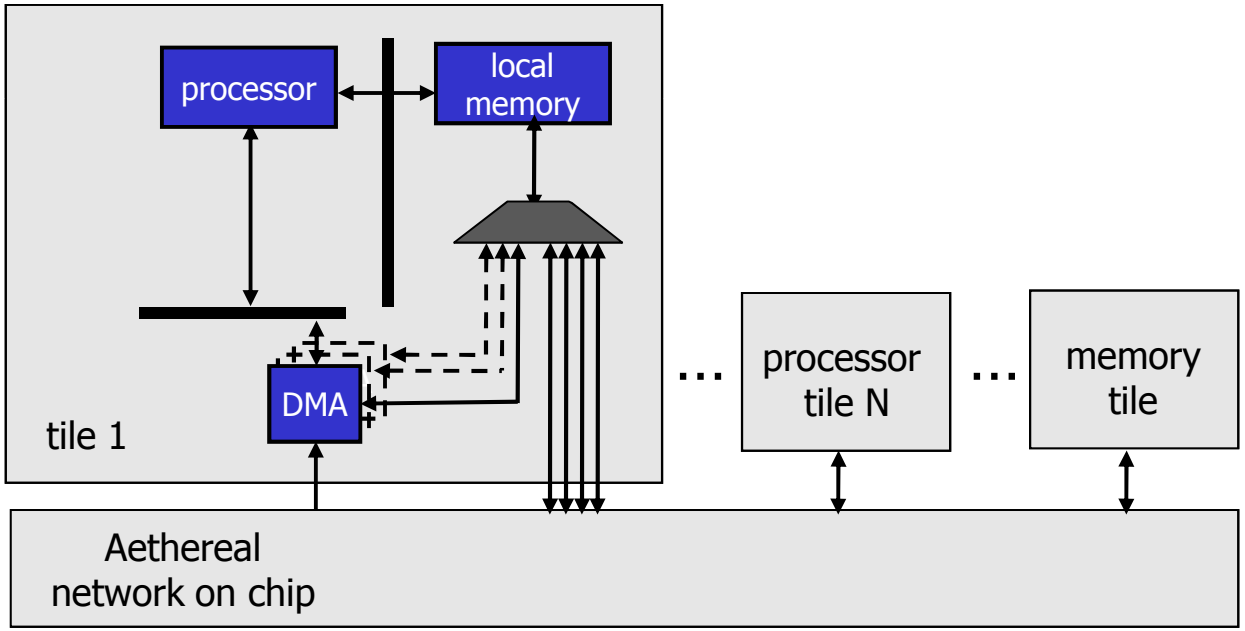
Processor-tile architecture (i)

- Dual-ported local memory (OK in FPGA)
  - otherwise large processor slowdown expected due to arbitration
  - the processor has 1 cycle access to local memory



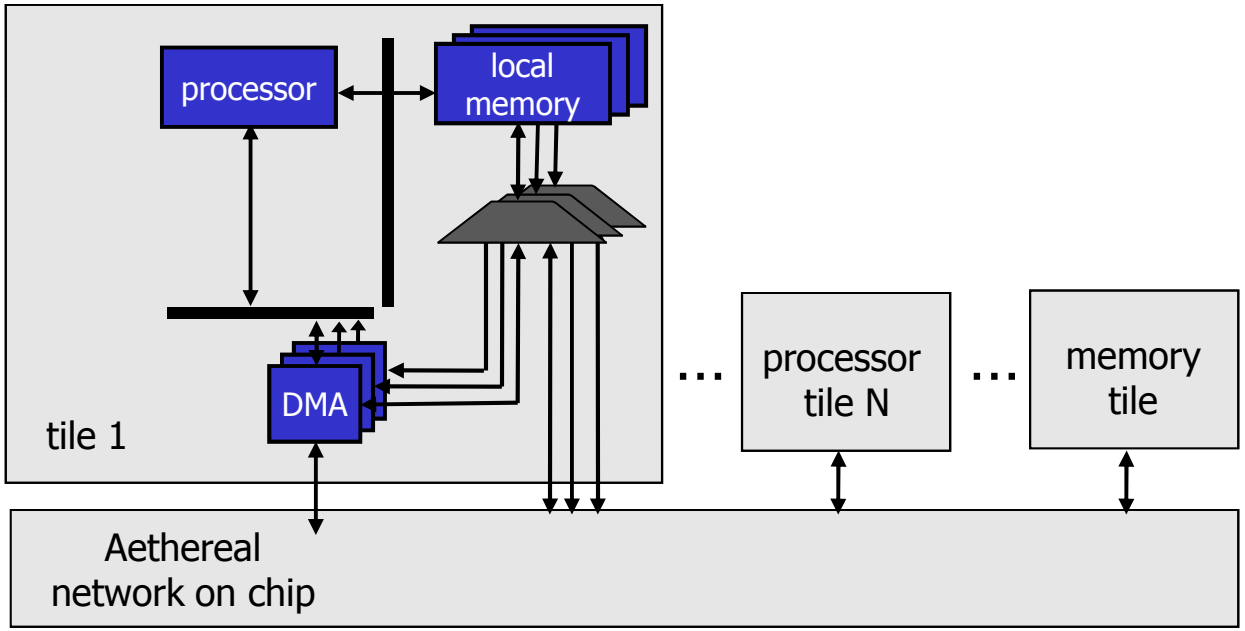
Processor-tile architecture (ii)

- Remind: several applications, composable sharing
  - composable, predictable arbitration between multiple connections and one or more DMA.



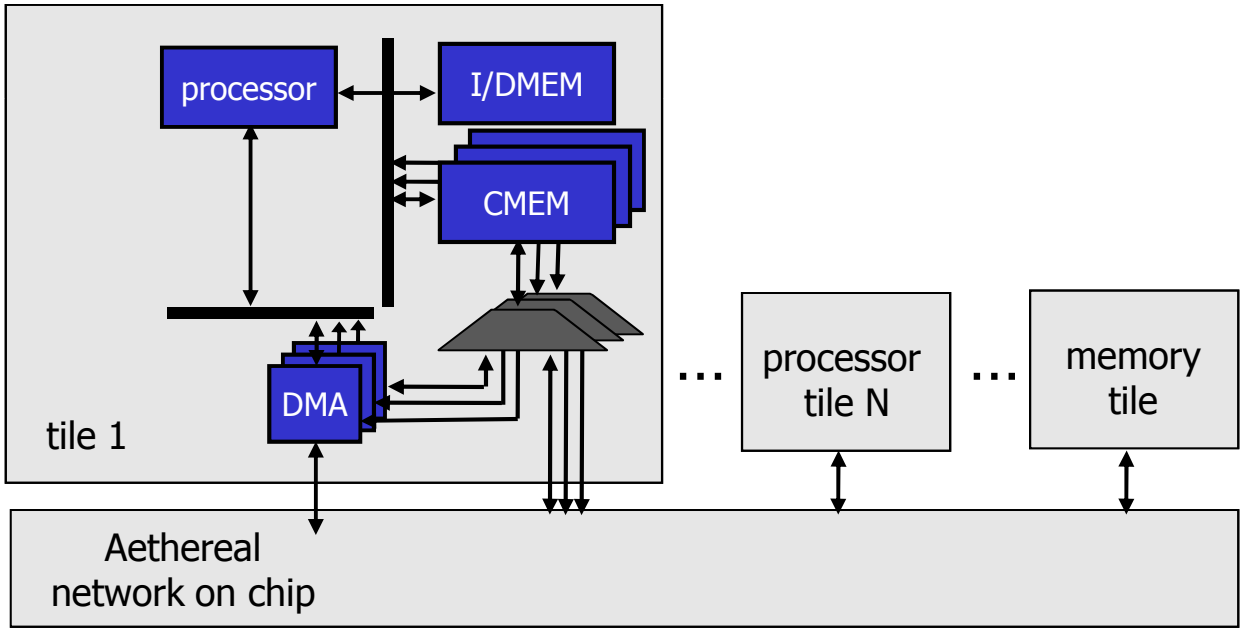
Processor-tile architecture (iii)

- or
  - 1 local memory & 1 DMA per application
  - predictable arbitration between NoC and DMA
  - problem: memory fragmentation



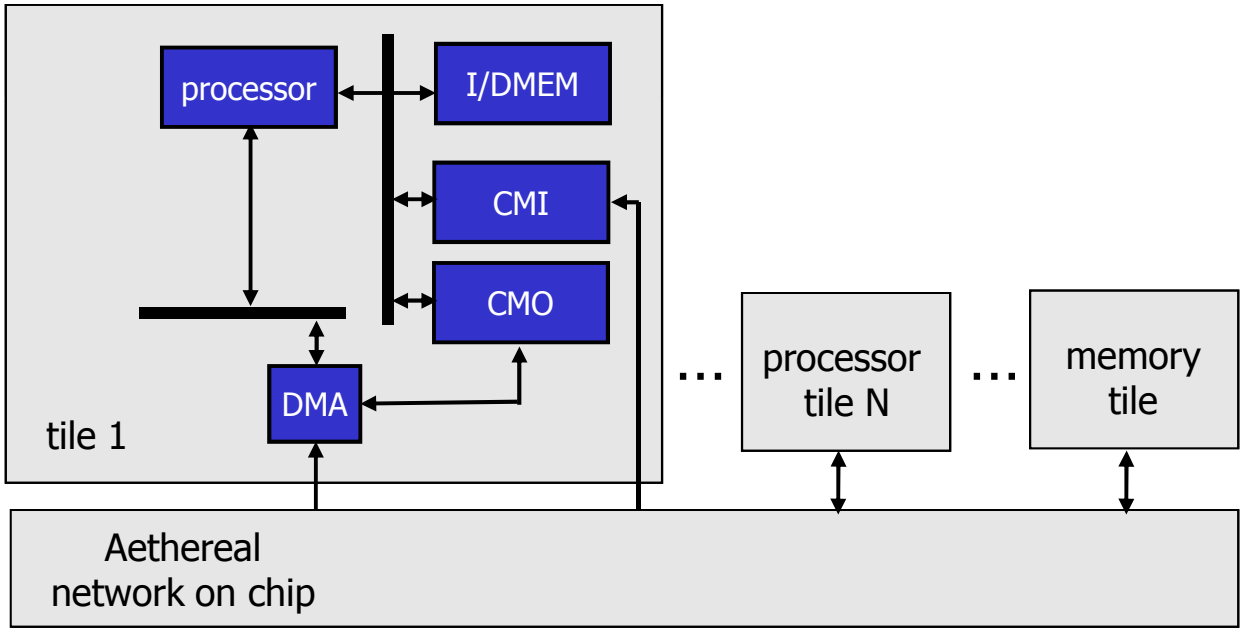
Processor-tile architecture (iv)

- shared DMEM and IMEM for all applications/tasks on the tile
- one CMEM per application arbitrated between DMA and NoC
- still some fragmentation, but less (typically CMEMs < DMEM)



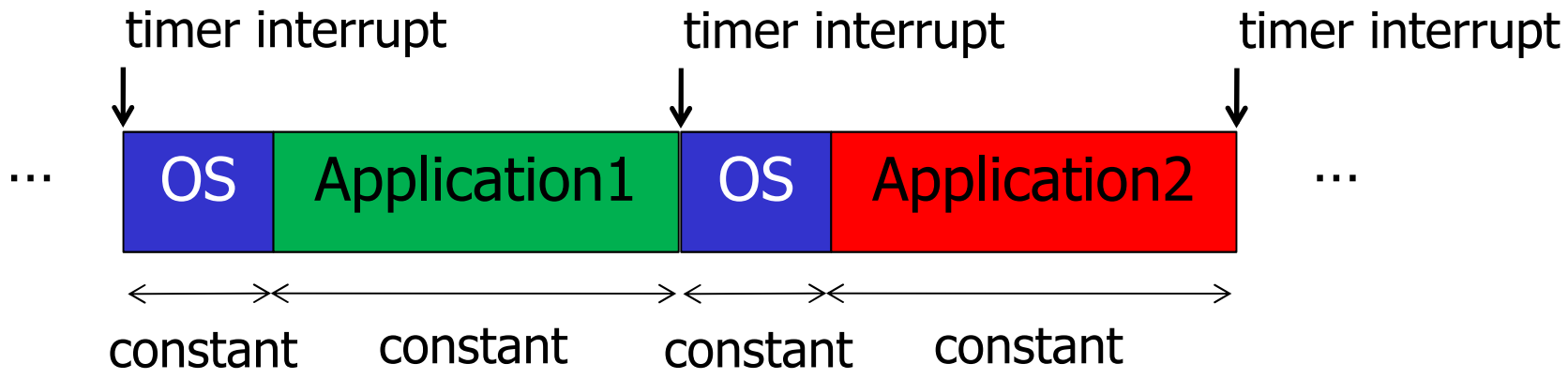
Processor-tile architecture (v)

- current local memory organization
- (at least one DMA, CMI, and CMO per application)



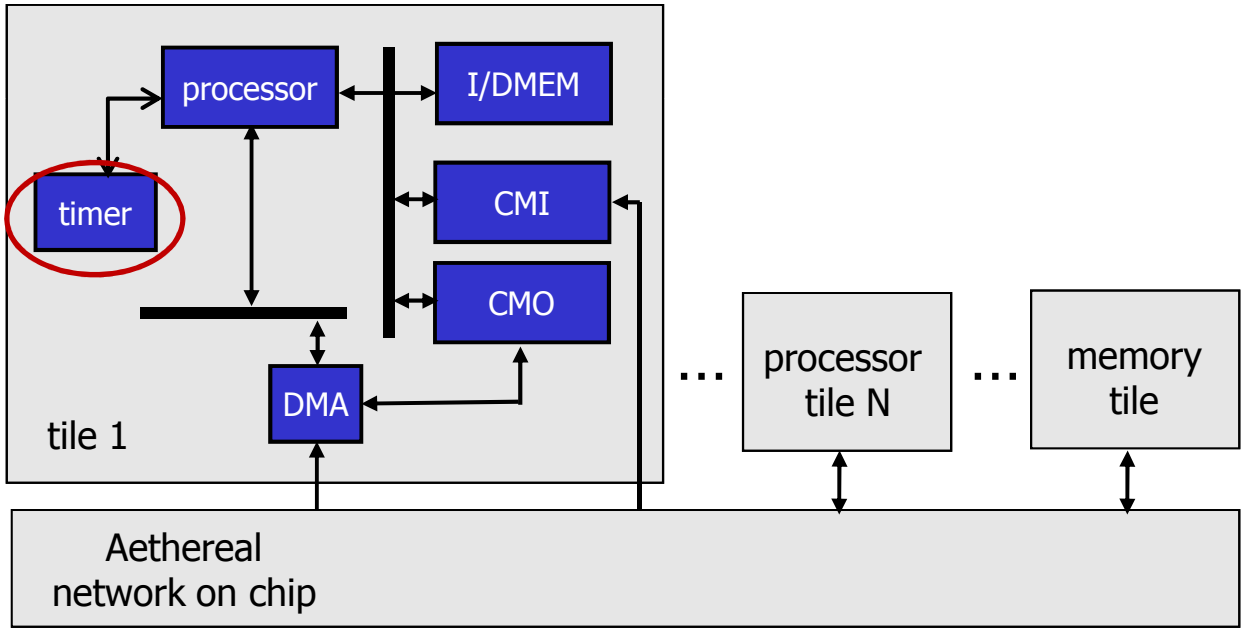
Processor sharing (i)

- compOSe (light-weight OS) on each processor
  1. schedules applications on the processor
  2. offers interfaces to the application
     - application management, task scheduling, FIFO communication, energy management



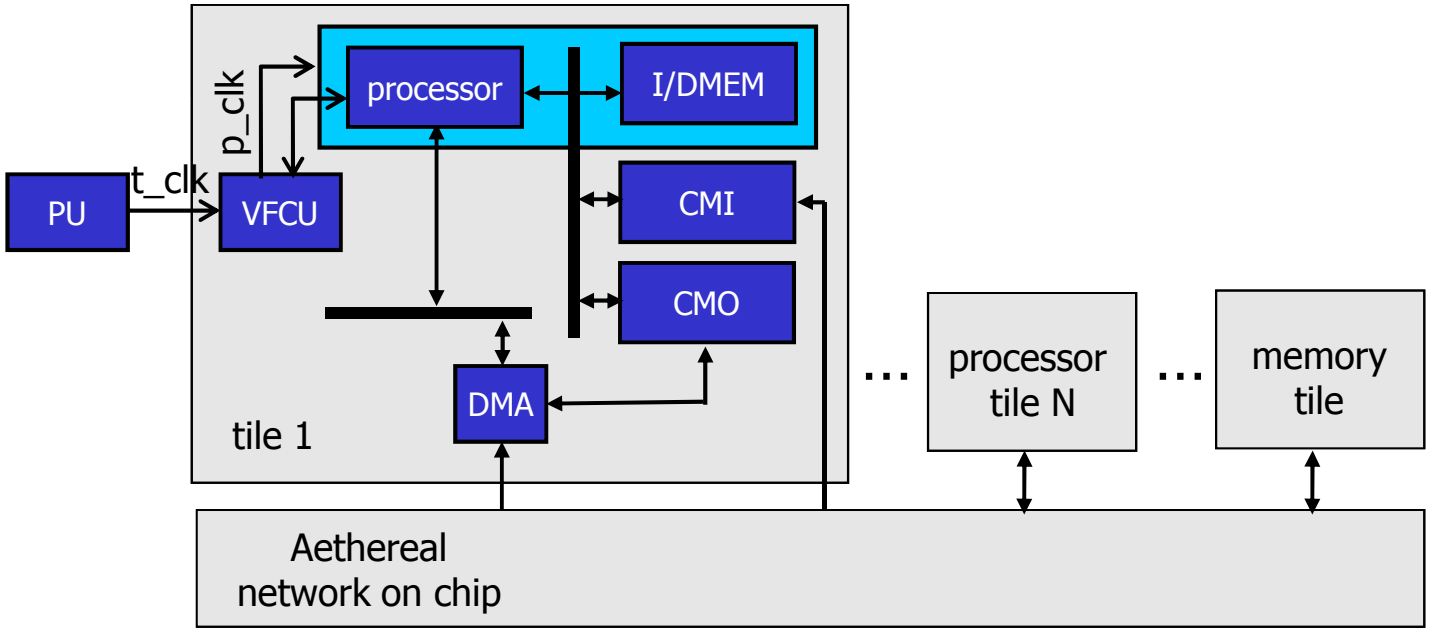
Processor sharing (ii)

- timer interrupts to trigger ComOSe and preempt applications
  - bounded preemption jitter is enough for predictability
  - interrupts at 'fixed' duration for composability
  - the only interrupts currently supported
    - on going work: virtualized interrupts.

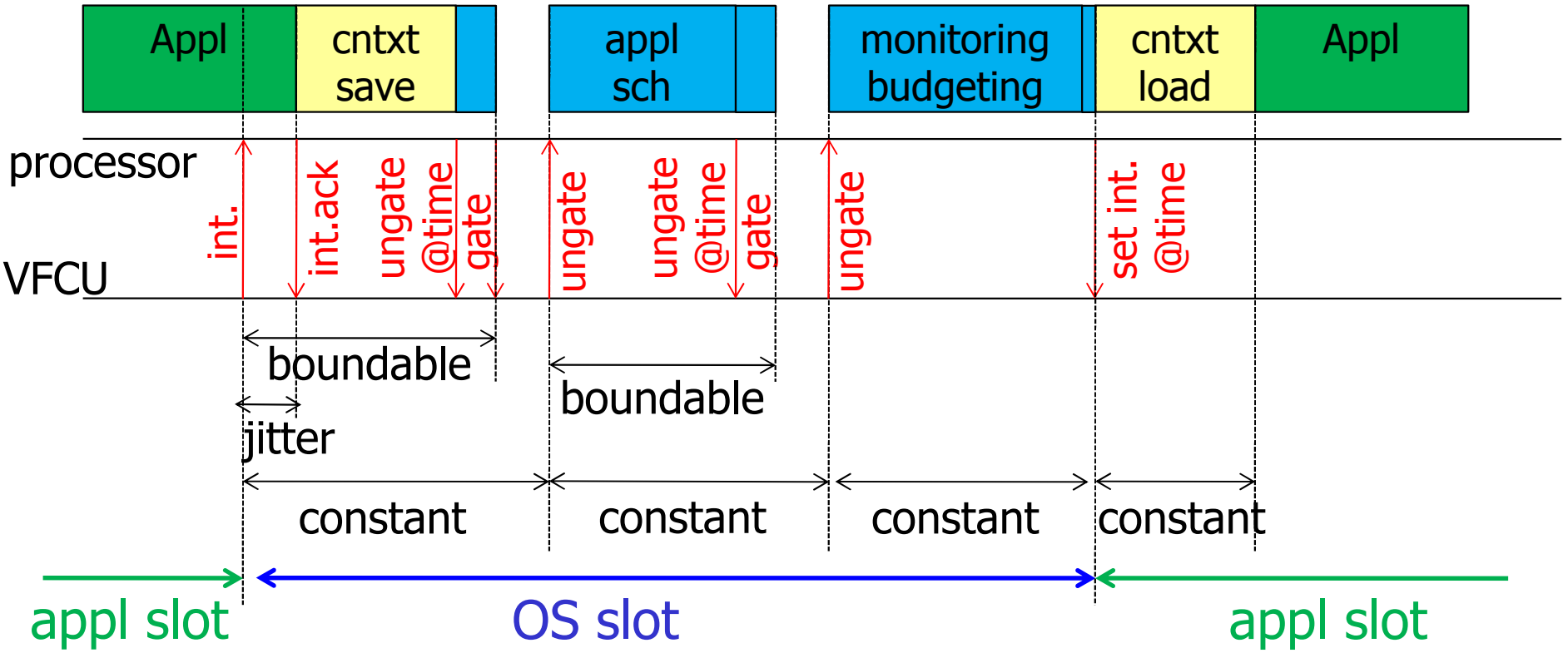


Parenthesis: frequency control

- in current implementation the timer is included in VFCU (clock and interrupts control unit)
  - can scale or gate/ungate the clock of the processors at fixed points in time
  - also manages timers



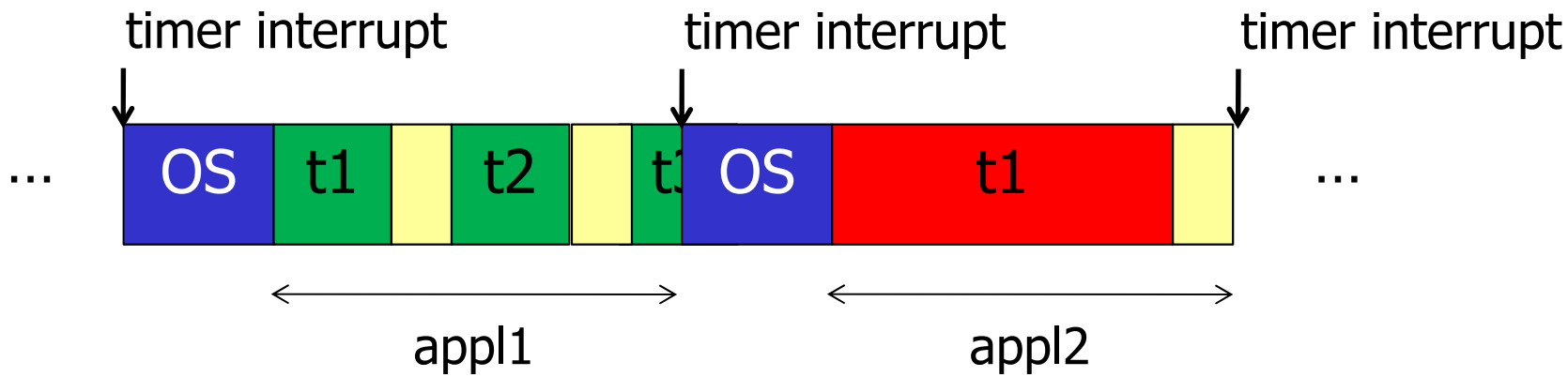
OS slot


Appl
cntxt save
appl sch
monitoring budgeting
cntxt load
Appl
processor
VFCU
int.
int.ack
ungate @time gate
ungate
ungate @time gate
ungate
set int. @time
boundable
boundable
jitter
constant
constant
constant
constant
appl slot
OS slot
appl slot

CompOSe scheduling

Two levels:

- composable arbitration inter-applications (which is a subset of budget-based, which in turn decouples application analysis).
- predictable arbitration intra-application (task scheduling) in application time, or OS time (deprecated).



Dataflow task and firing

Dataflow-friendly task

```
while (1)
// firing rules check
   if (firing_rule()) {
    read FIFO1
    read FIFO2
    …
     // task computation
    task_func(…)
    write FIFO3
    write FIFO4
    …
   }
```

Dataflow task execution

Dataflow-friendly task

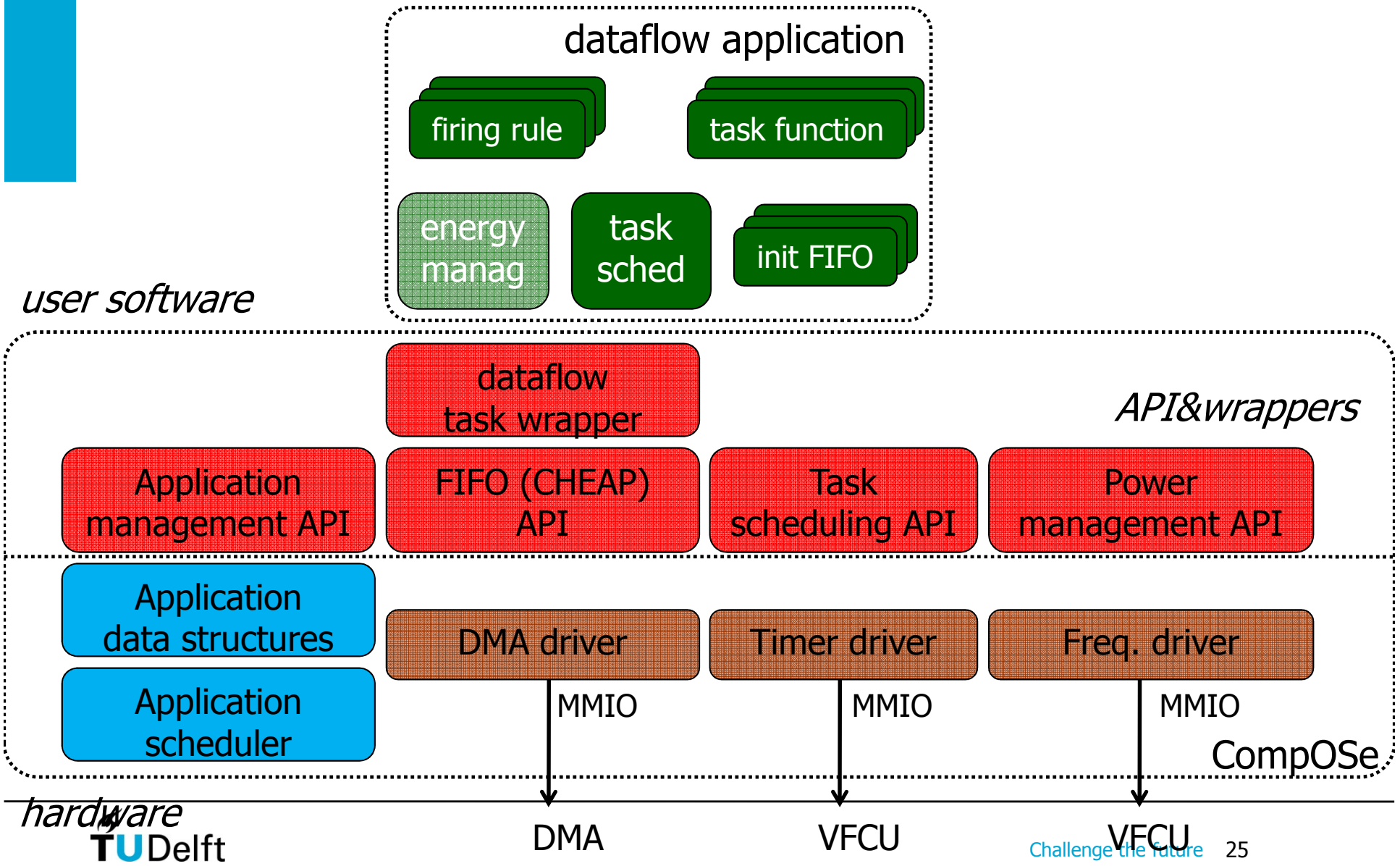
`while (1)`	Wrapper provided by CompOSe
// firing rules check `if (firing_rule()) {`	In task scheduler Provided by application designer

`read FIFO1`
`read FIFO2`
…
	Wrapper provided by CompOSe

// task computation `task_func(in, out)`	Provided by application designer

`write FIFO3`
`write FIFO4`
…
`}`
	Wrapper provided by CompOSe

CompOSe interfaces (overview)



Application management API

- add/remove of applications, tasks, FIFOs.
- `os_add_application(APP_ID, NBR_TASKS, NBR_FIFOS, (task_scheduler_callback) task_sched, param_task_sched);`
- `os_add_task(ID, APP_ID, ..., (task_callback) task_func, (firing_rule_callback) firing_rule, ...);`
- `os_add_fifo(ID, ..., LWC, LRC, RWC, RRC, PROD_BUF, DATA_BUFF, ...);`

- application management API called:
  - statically, at application initialization (privileged code)
  - dynamically, by a System Application that loads other applications at run-time (on going work).

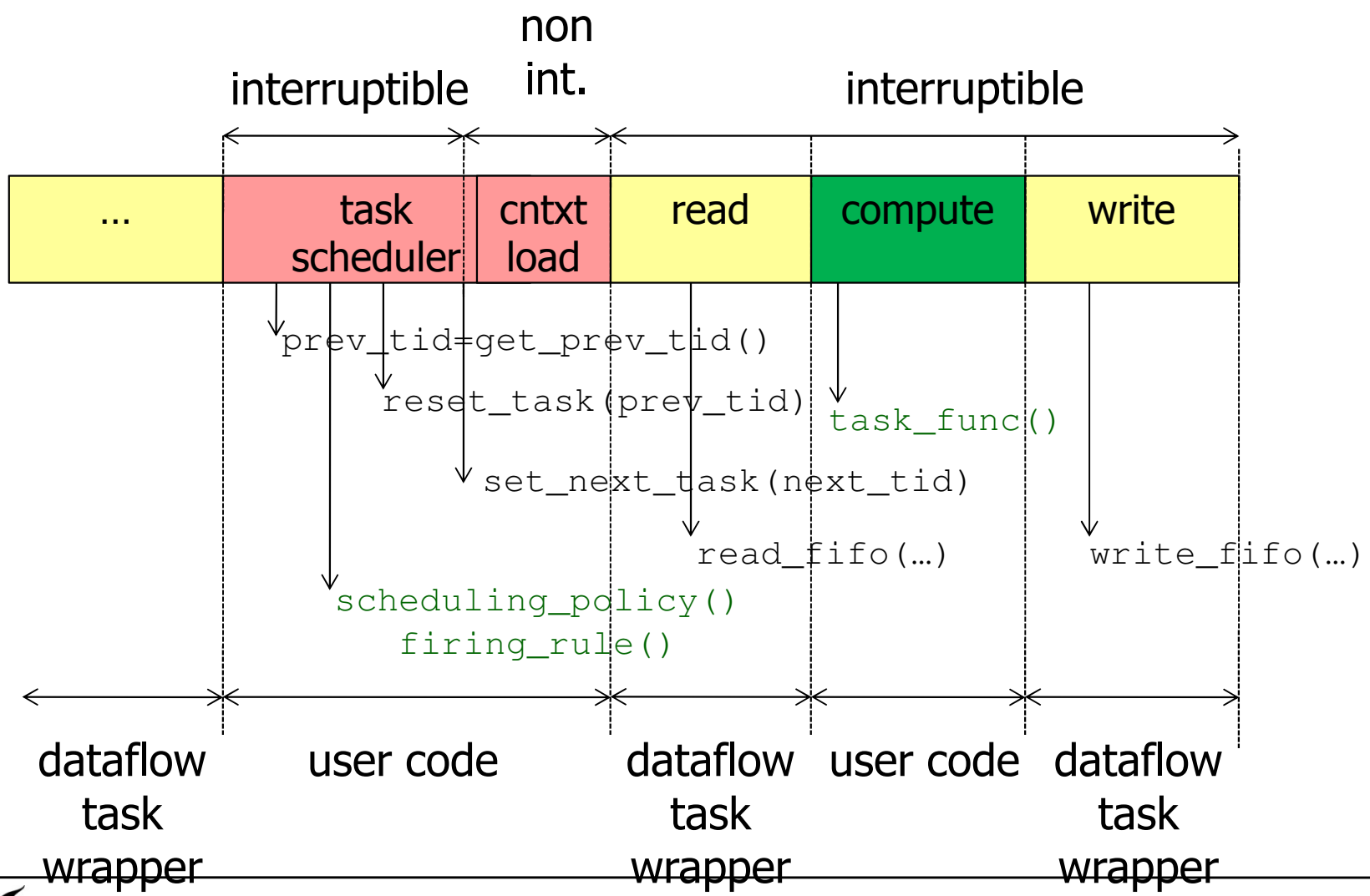
FIFO API

- during application initialization / init FIFO:
  - `write_initial_tokens(id);`
- during task execution:
  - `read_fifo(int id, int nbr_tokens, int* buffer);`
  - `write_fifo(int id, int nbr_tokens, int* buffer);`
  - called by the dataflow task wrapper

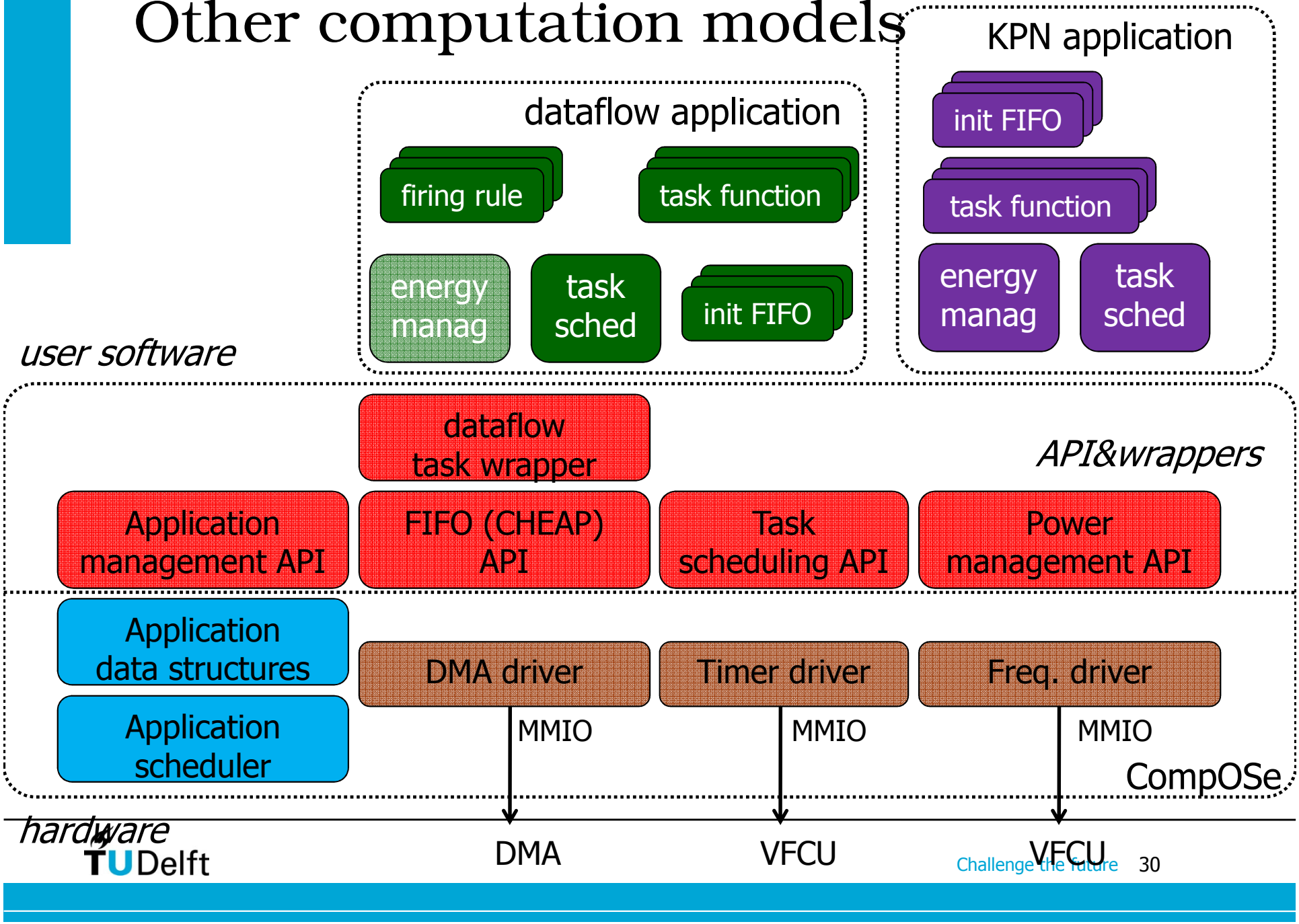
Task scheduling API

- during task scheduler:
  - `int get_prev_task_id()`
  - `void reset_task(int id)`
  - `void set_next_task(int id)`
- currently cooperative task scheduling
  - task wrapper calls the task scheduler after each task iteration
- preemptive task scheduling (work in progress)

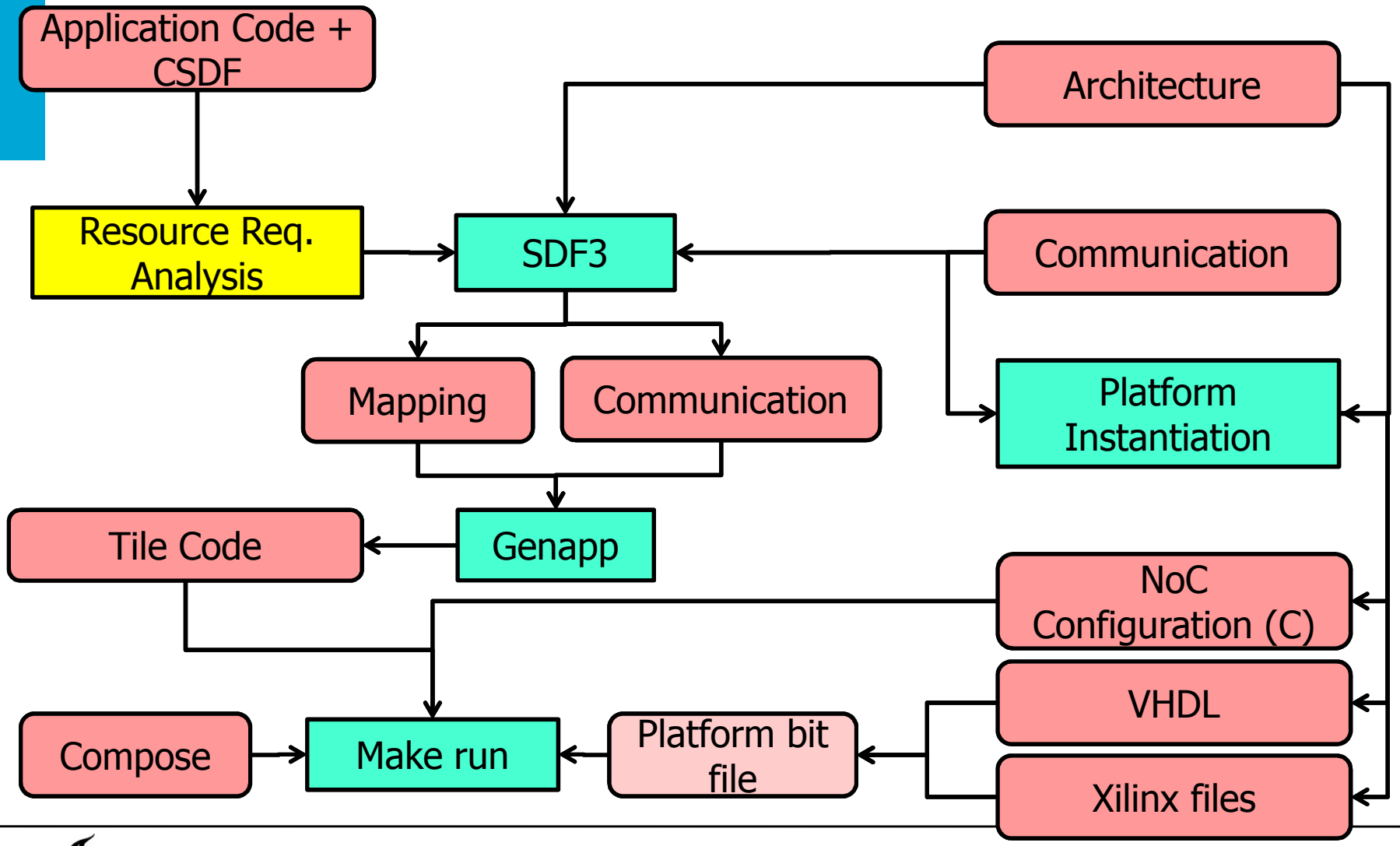
Application slot



Other computation models



CompSoC/SDF3 flow



Conclusions

- predictable, composable architecture
- CompOSe
  - implements processor time sharing
    - composable between applications
    - predictable within an application
  - implements APIs
    - develop dataflow, KPN, sequential C applications
    - energy and power management per application
- automatic flow to generate hardware and software for FPGA prototype
  - for FRT includes SDF3 for dataflow analysis
  - for all MOCs, from user application
    - provide application, mapping, architecture, communication