Latency-preserving software pipelining of predicated reservation tables for distributed hard real-time applications

Thomas Carle – Dumitru Potop-Butucaru
INRIA Paris-Rocquencourt, FRANCE
Team AOSTE

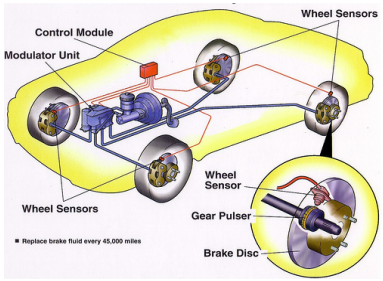
Outline

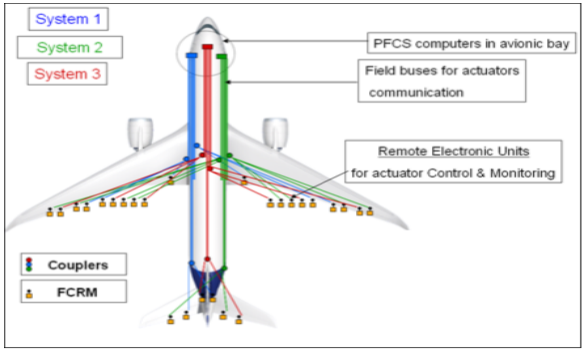
- Throughput optimization problem
- Previous work (software pipelining)
- System models (pipelined and non-pipelined)
- Pipelining algorithms
- A complex example
- Conclusion and future work

Application areas

Complex embedded control applications:

- Cyclic, periodic execution
- Safety-critical applications
  - Hard Real-Time constraints
  - Focus on functional and temporal correctness
- Distributed implementations





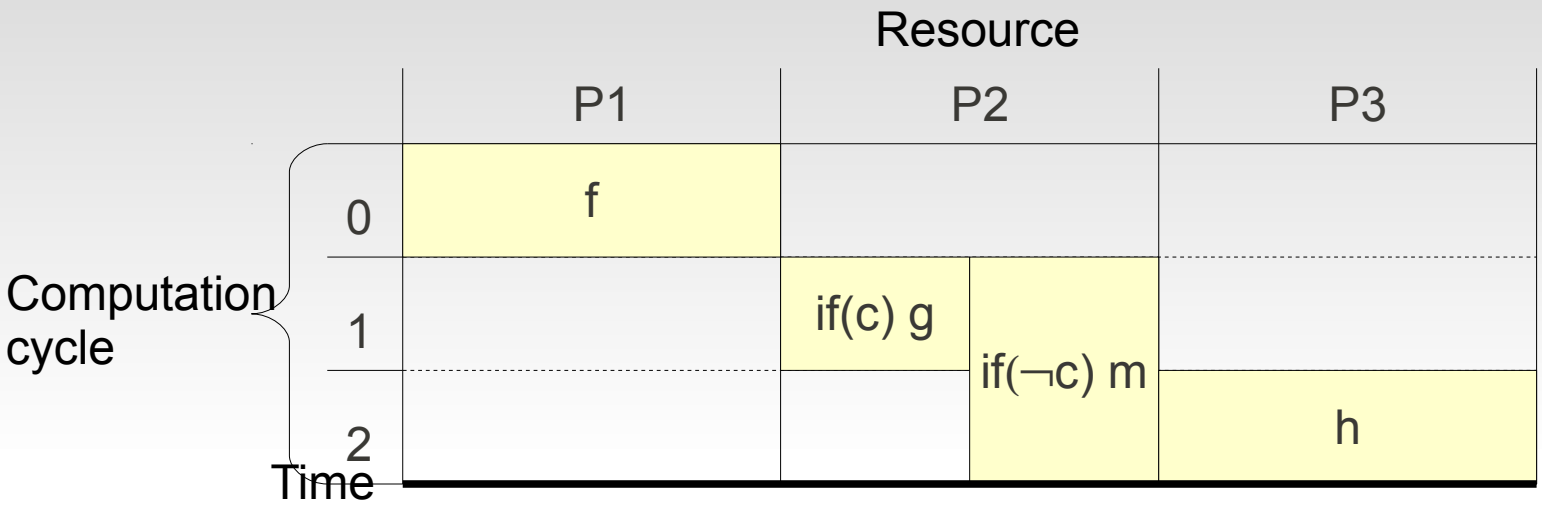
Static scheduling

Our work focuses on static schedules: scheduling/reservation tables

Validated by industrial standards: **ARINC 653, AUTOSAR, FlexRay,**...

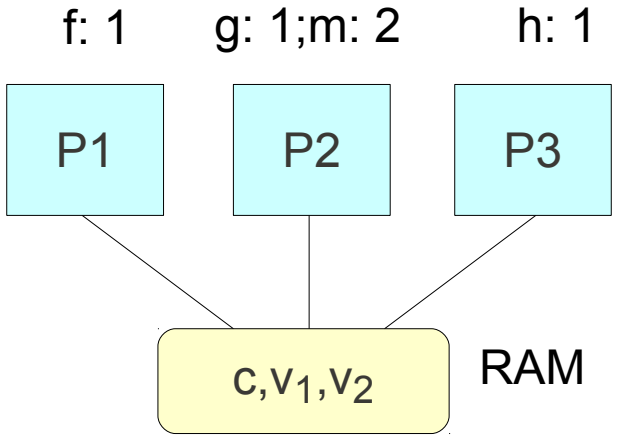
Defines one cycle of execution, repeated **periodically**

Motivating example

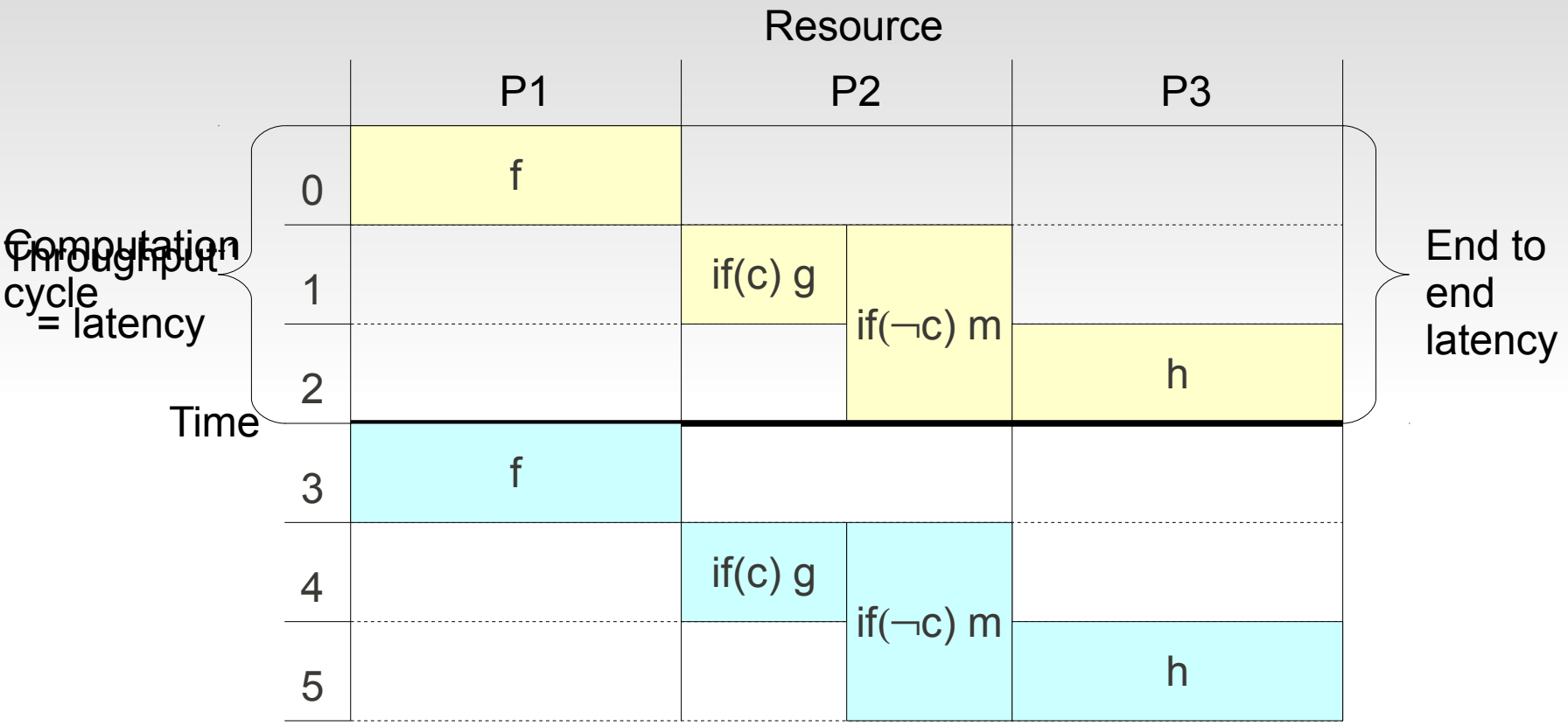


Code:

v_2:=v_2_init;
loop
(v_1,c)=f(v_2);
if c **then** v_2:=g(v_1)
else m(v_1);
h(v_2);
end



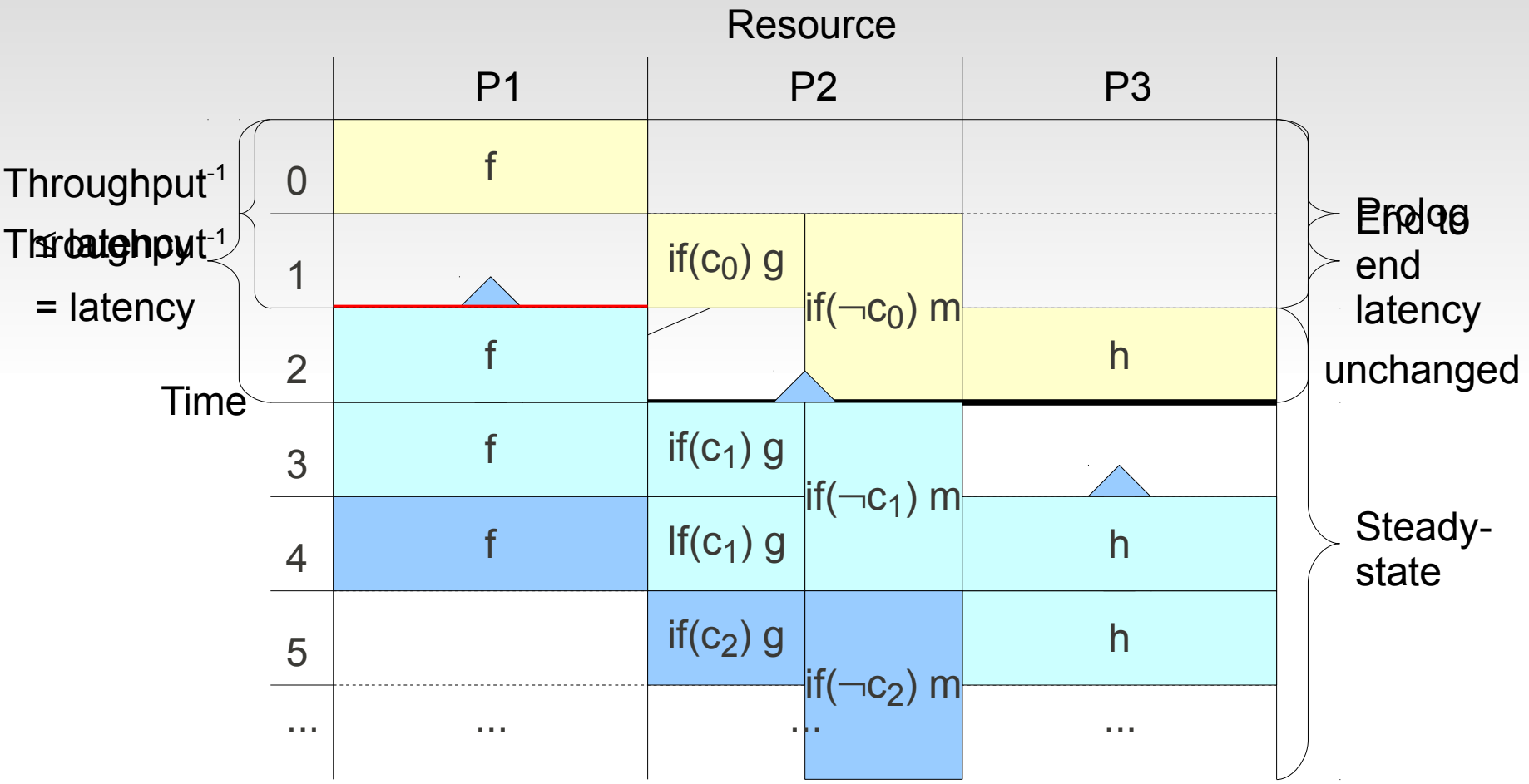
Motivating example



Latency: number of time units between the beginning and the end of the execution of a cycle

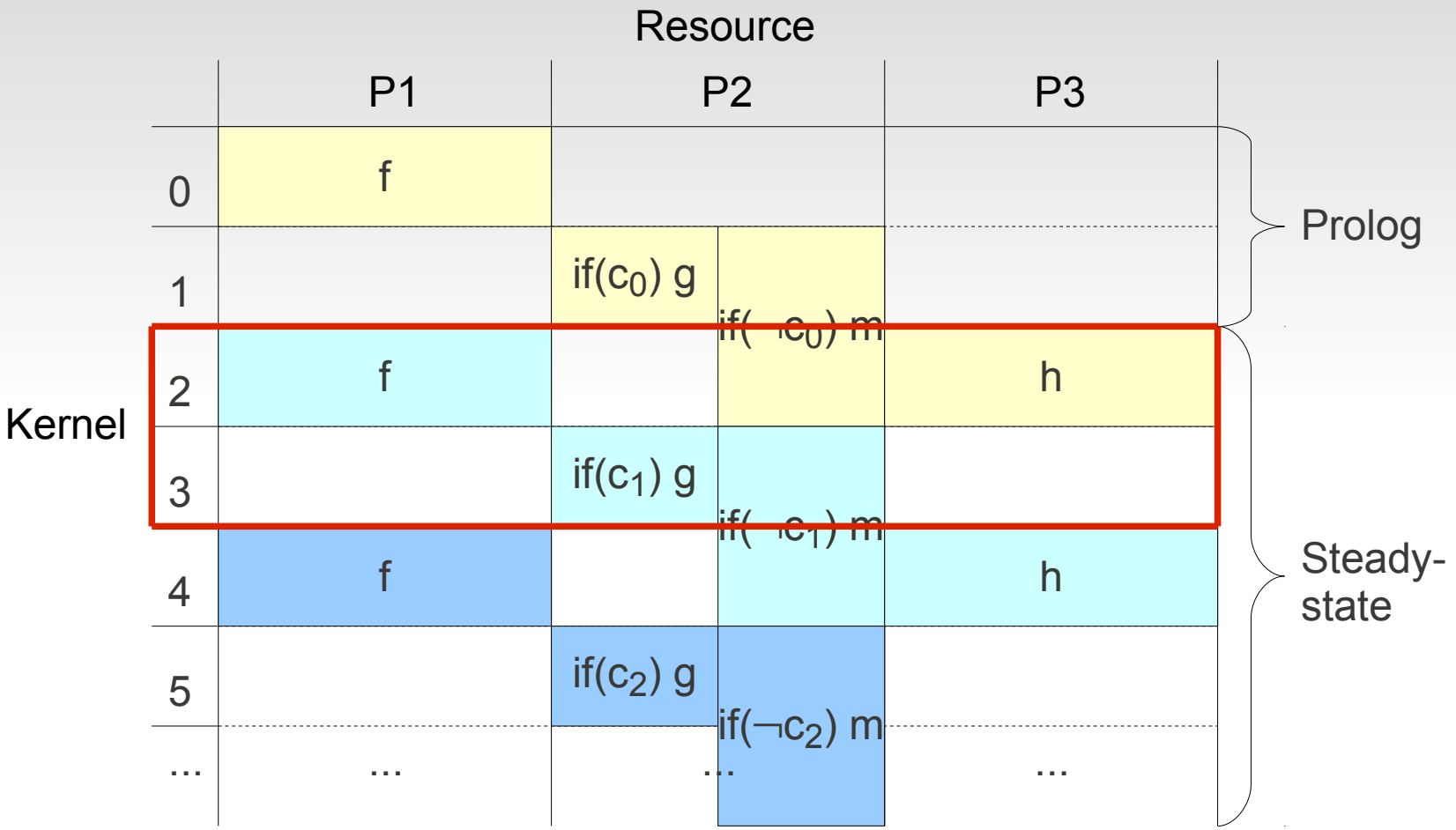
Throughput: number of cycles executed in one time unit

Our objective



Goal : increase throughput while keeping the system's latency, I/O function, and periodic behaviour

Our objective



Prolog and steady-state are instances of the kernel

Previous work(1): Software Pipelining

Scheduling techniques for **parallelizing loop computations**:

- Developped since the 1980's,
- First aimed at massively parallel architectures such as **VLIW** and superscalar machines,
- Now common optimization, present in most **compilers**,
- Similar to hardware pipelining: out-of-order execution,
- Reordering done in the compiler instead of in the processor.

Software Pipelining vs our work

- Low-level vs coarse-grain code generation technique
- Goal : optimize average-case throughput by reorganizing operations order to take advantage of parallelism vs optimize worst-case throughput without degrading the cycles latency by preserving the intra-cycle scheduling
- No periodicity for applications with data-dependent control vs preservation of the periodic behaviour of the application
- Low degree of control over operators/functional units for conditional execution vs exploitation of conditional execution to improve the pipelining process

Previous work(2): Retiming

- Optimization method in which **registers** in a synchronous circuit are **relocated** in order to **improve the throughput** or memory consumption of an application,

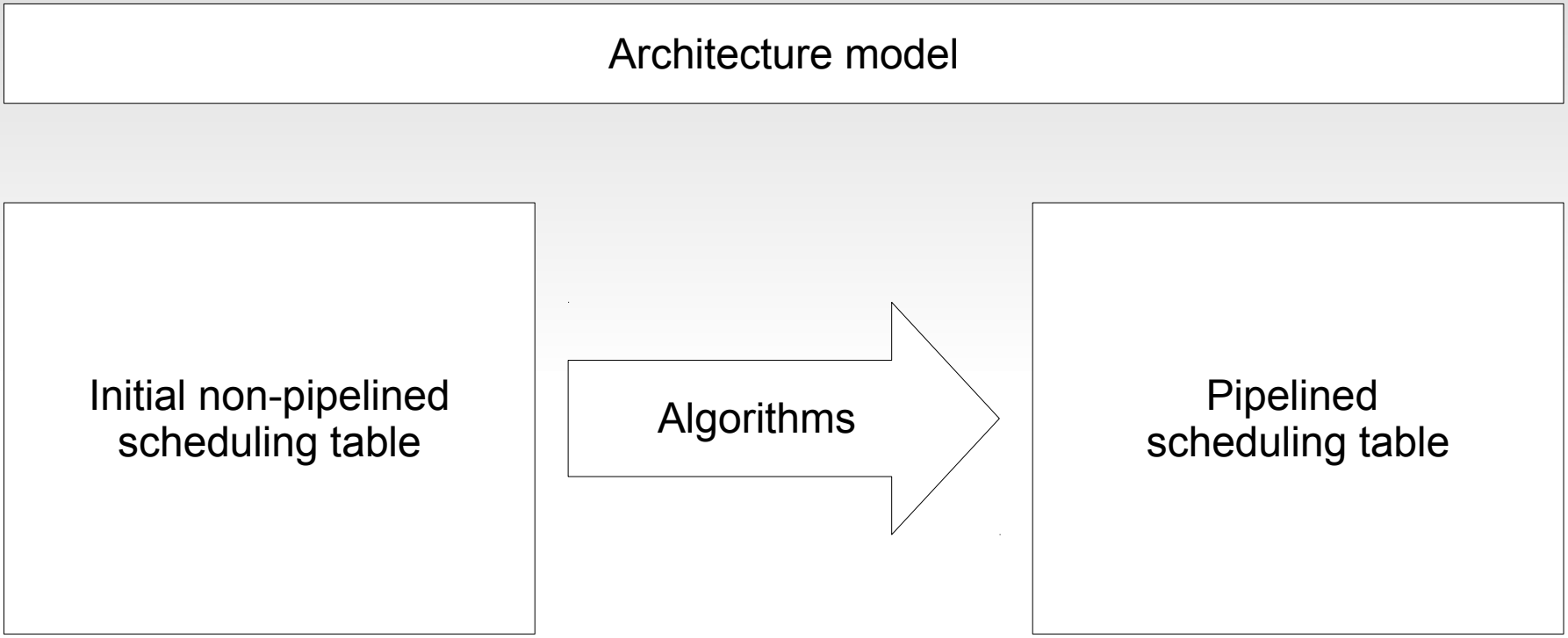
- Very similar to our techniques e.g. **no increase in latency** after applying the retiming techniques, preservation of the **I/O function**,

- Nevertheless: **no support for conditional execution/predication**.

Previous work(3): Real-Time Software Pipelining

- Builds a pipelined schedule for the application,
- Demonstrated on e.g. multimedia streaming applications,
- Again, no optimization for conditional execution

Elements of our approach



We design low-level implementation models that can be integrated at the end of the development cycle

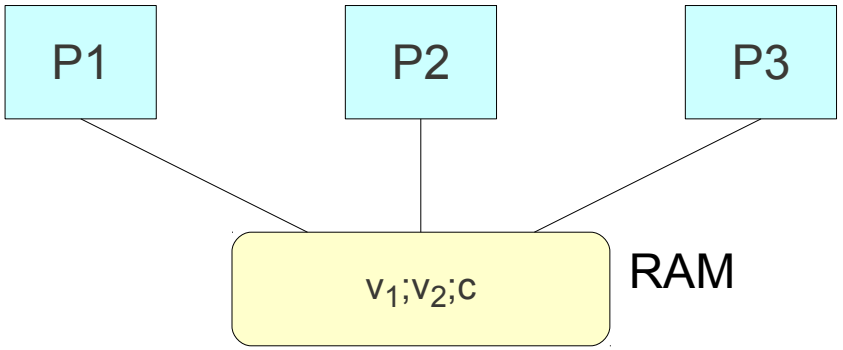
Architecture model

Bipartite undirected graph: **A=<P,M,C>**, where:

- P: "processors", i.e. computation and communication resources capable of independent execution (Processors, DMAs, ...),
- M: RAM blocks,
- (P,M) $\in$ C indicates that processor P has direct access to memory block M.

RAM blocks: sets of disjoint untyped memory cells

Example:



Reservation/Scheduling table

S=<p,O,Init>, where :

- p : activation period of execution cycles, equal to the length of the reservation table,
- O : Set of scheduled operations,
- Init : set of initial values of all memory cells (can be *nil* or a constant).

Reservation/Scheduling table

Scheduled operation o:

- In(o): set of **memory cells** whose data is used as **input** by o,
- Out(o): set of **memory cells written** by o,
- Guard(o): **execution condition of o** (predicate over memory cells),
- Res(o): **set of "processors"** used during the execution of o,
- t(o): **start date** of o,
- d(o): **duration** of o, maximum time budget which can be ensured throught WCET analysis.

Reservation/Scheduling table

Well-formed properties:

- Exclusive resource use: for O_1, O_2 scheduled on the same resource, if Guard(O_1) $\wedge$ Guard(O_2) $\neq$ false, then $t(O_1) \geq t(O_2)+d(O_2)$ or $t(O_2) \geq t(O_1)+d(O_1)$,
- No data races : if O_1 writes variable v_1 and O_2 uses (reads or writes) v_1, then $t(O_1) \geq t(O_2)+d(O_2)$ or $t(O_2) \geq t(O_1)+d(O_1)$, or Guard(O_1) $\wedge$ Guard(O_2) = false,
- Causal correctness.

Enough to describe non-pipelined schedules.

Pipelined Reservation/Scheduling table

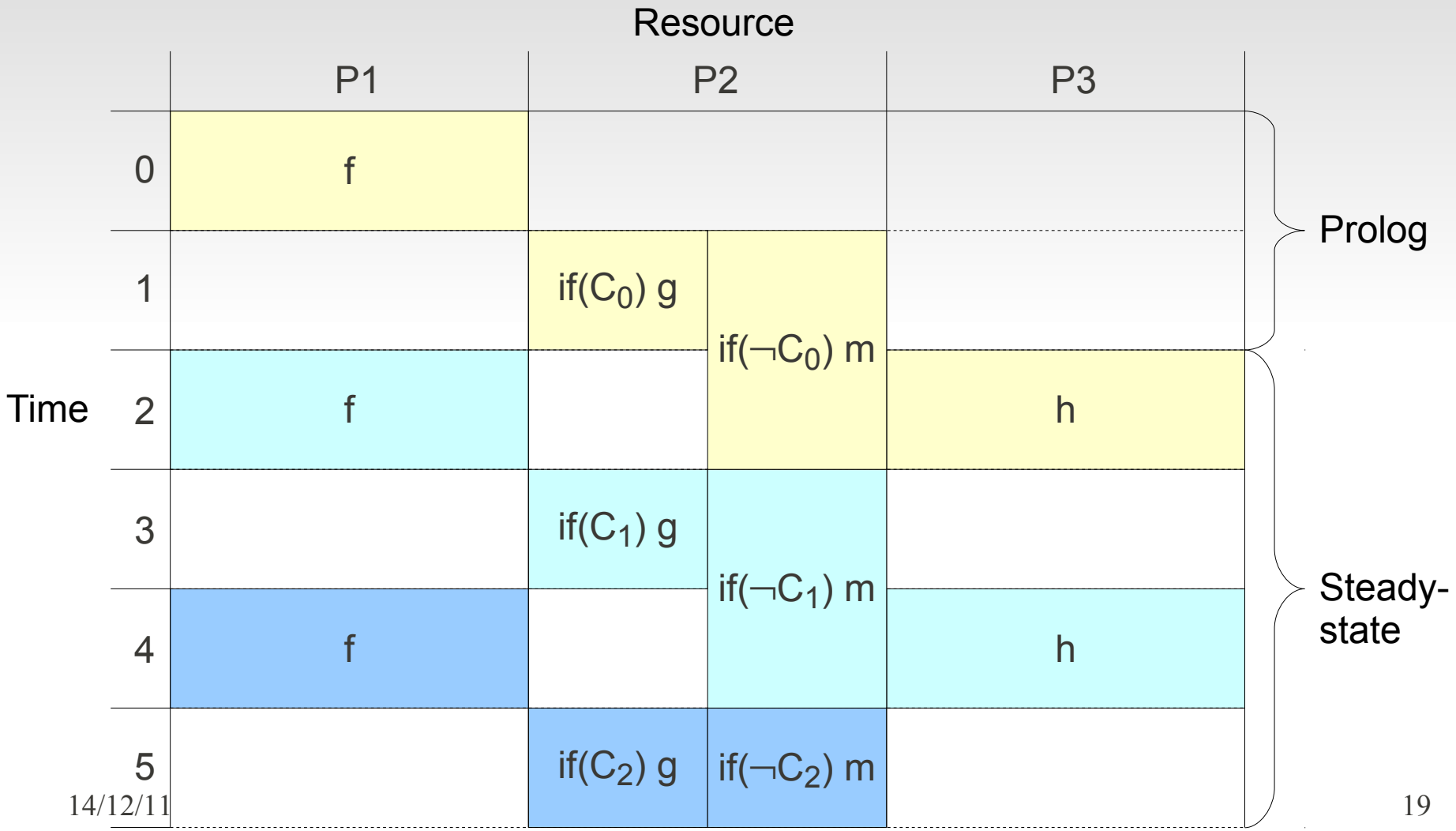
For pipelined schedules, each scheduled operation o also has a **start index** *fst(o).* It accounts for the prologue phase, where operations progressively start to execute.

If operation o has fst(o)=n, it will first be executed in the **pipelined cycle** of index n.

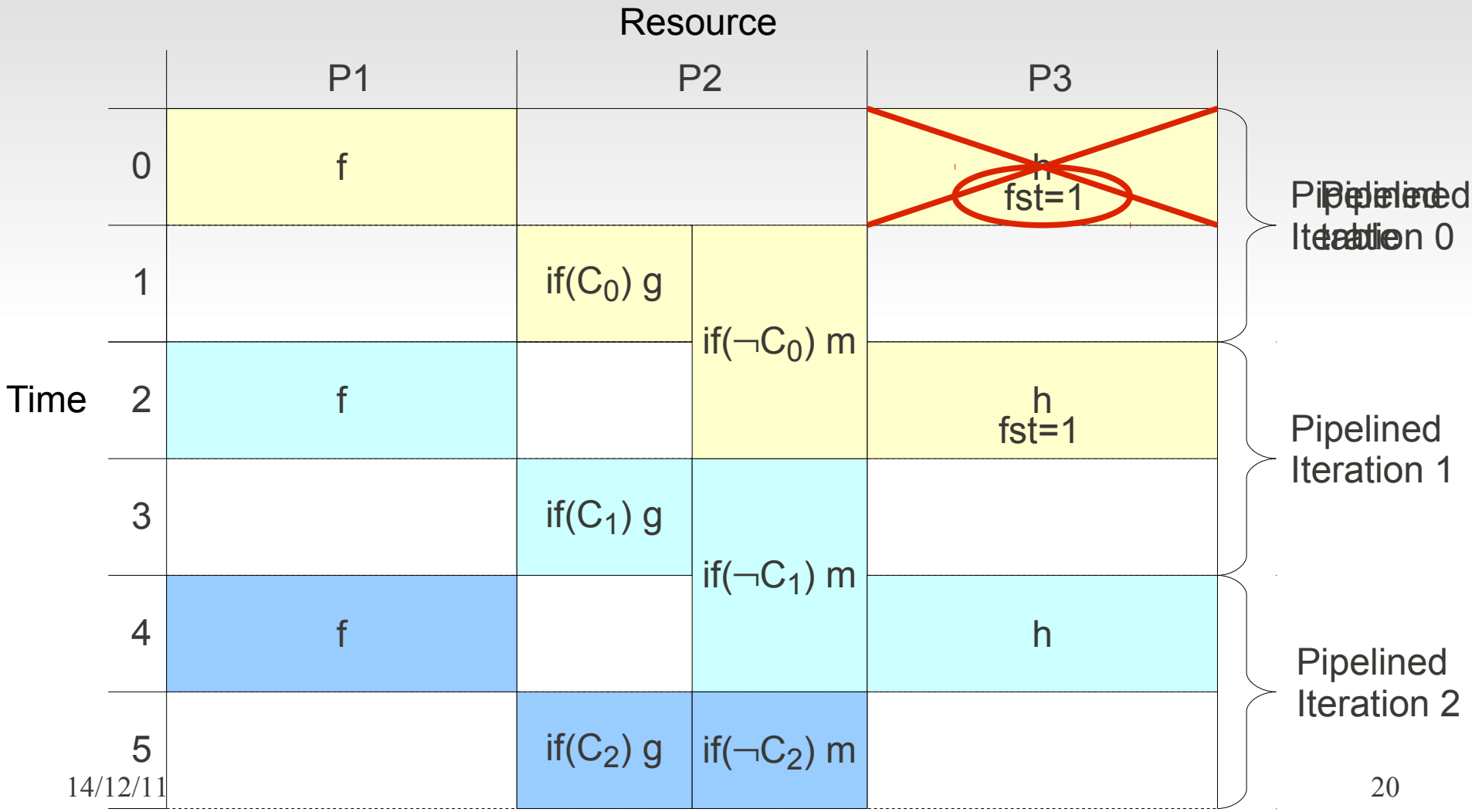
Due to periodicity, the description of the schedule of the kernel with start indexes is enough to describe the whole execution of the system.

Memory elements can be modified to take into account the variable replication process (described later)

Pipelined Reservation/Scheduling table



Pipelined Reservation/Scheduling table



Pipelining algorithm

- Constraints:
  - need to respect inter-cycle data dependency
  - no two operations can use a "processor" at the same time
  - no memory cell can be written by an operation and used (written or read) by another at the same time

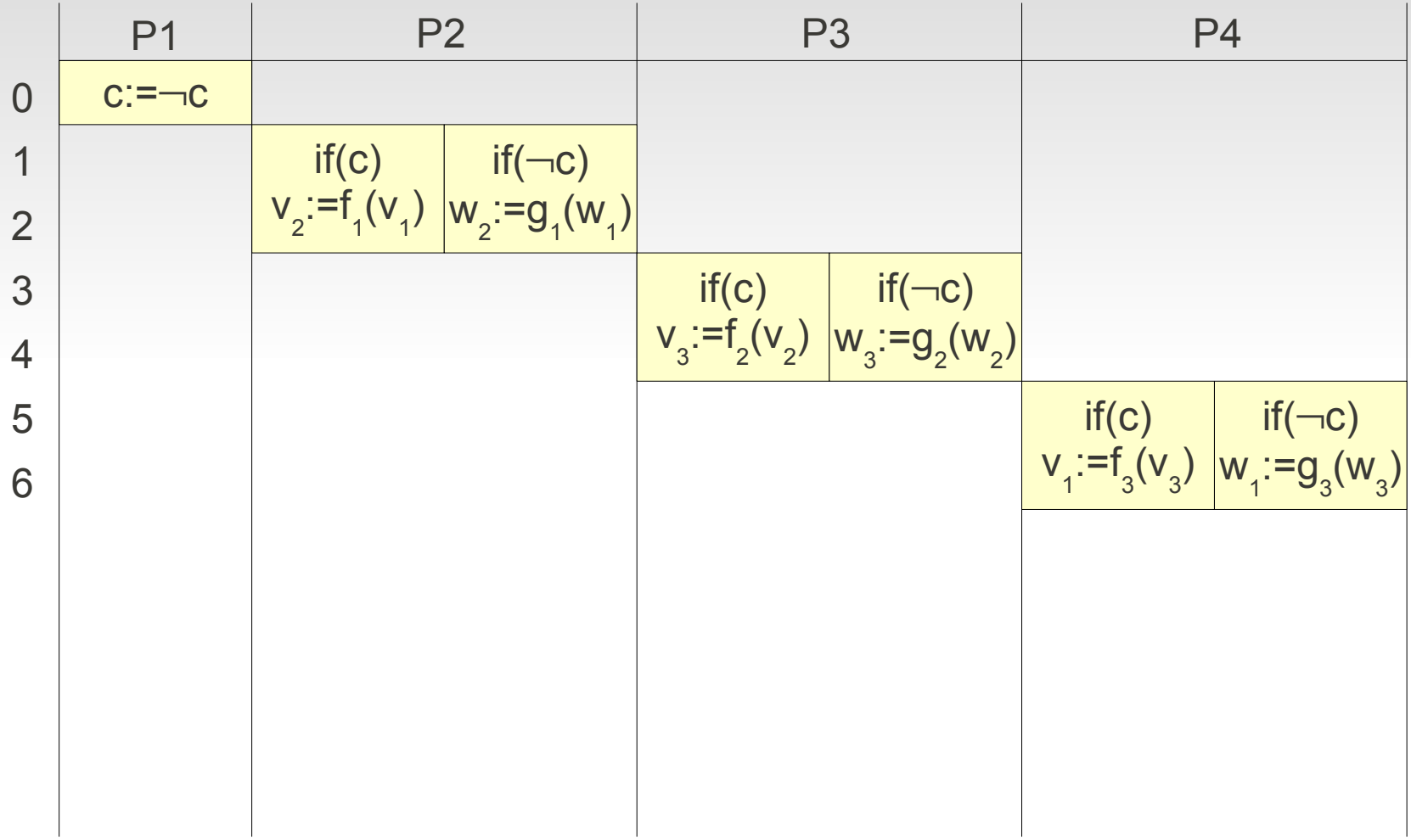
- Our algorithm
  - Enforces the fulfilment of these constraints
  - Incrementally builds the Data Dependency Graph of the application
  - Takes advantage of guards during pipelining (better than existing work)
  - Specific memory handling

Pipelining algorithm

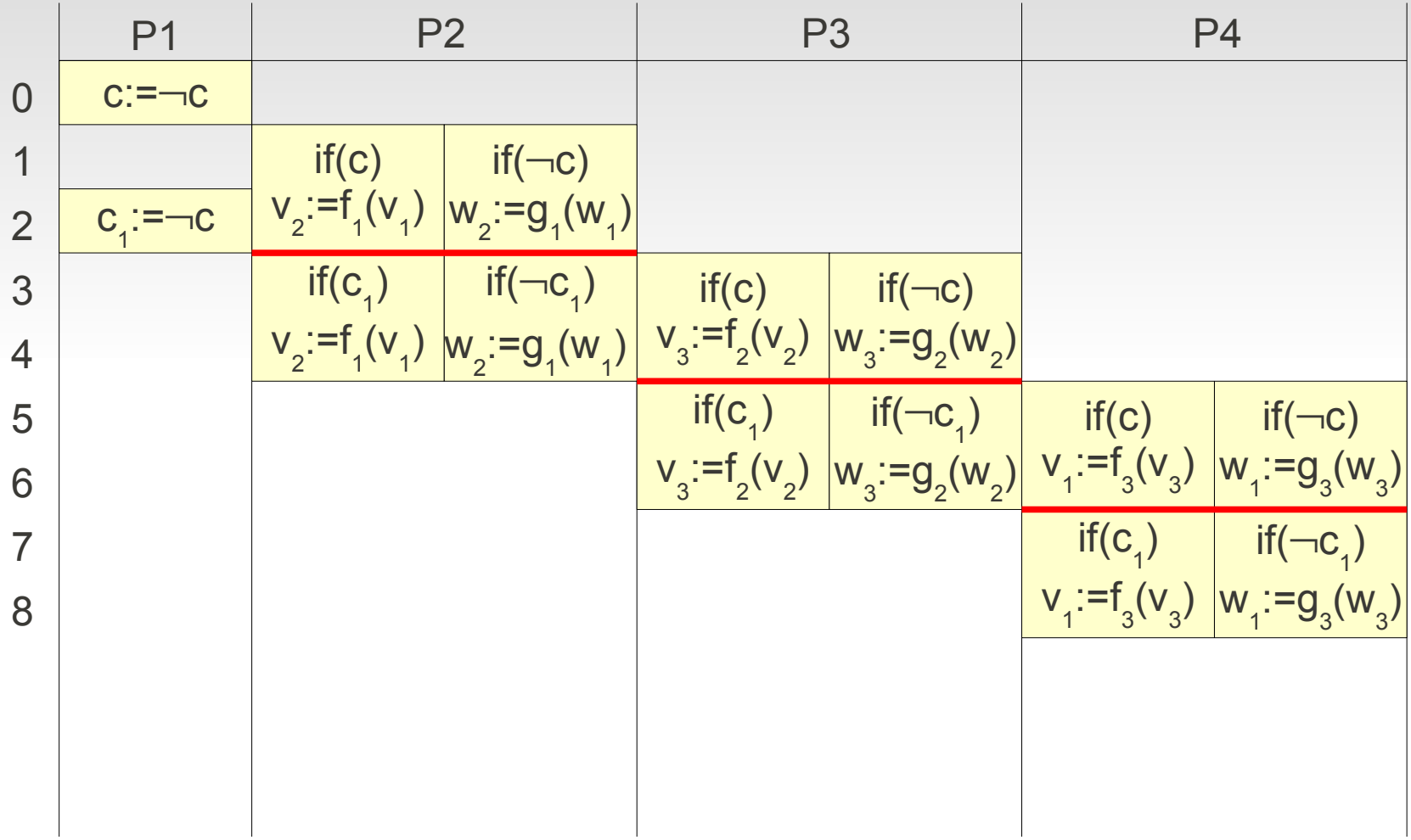
- Relies on the incremental construction of the **Data Dependency Graph (DDG)** of the application i.e. the set **{(o_1,o_2,n)}** for all o_1 and o_2 such that **$In(o_2) \cap Out(o_1) \neq \varnothing$**, and o_1 **happens n cycles** before o_2,

- Uses an **SSA transformation** before performing a symbolic execution of the different iterations in order to construct the DDG.

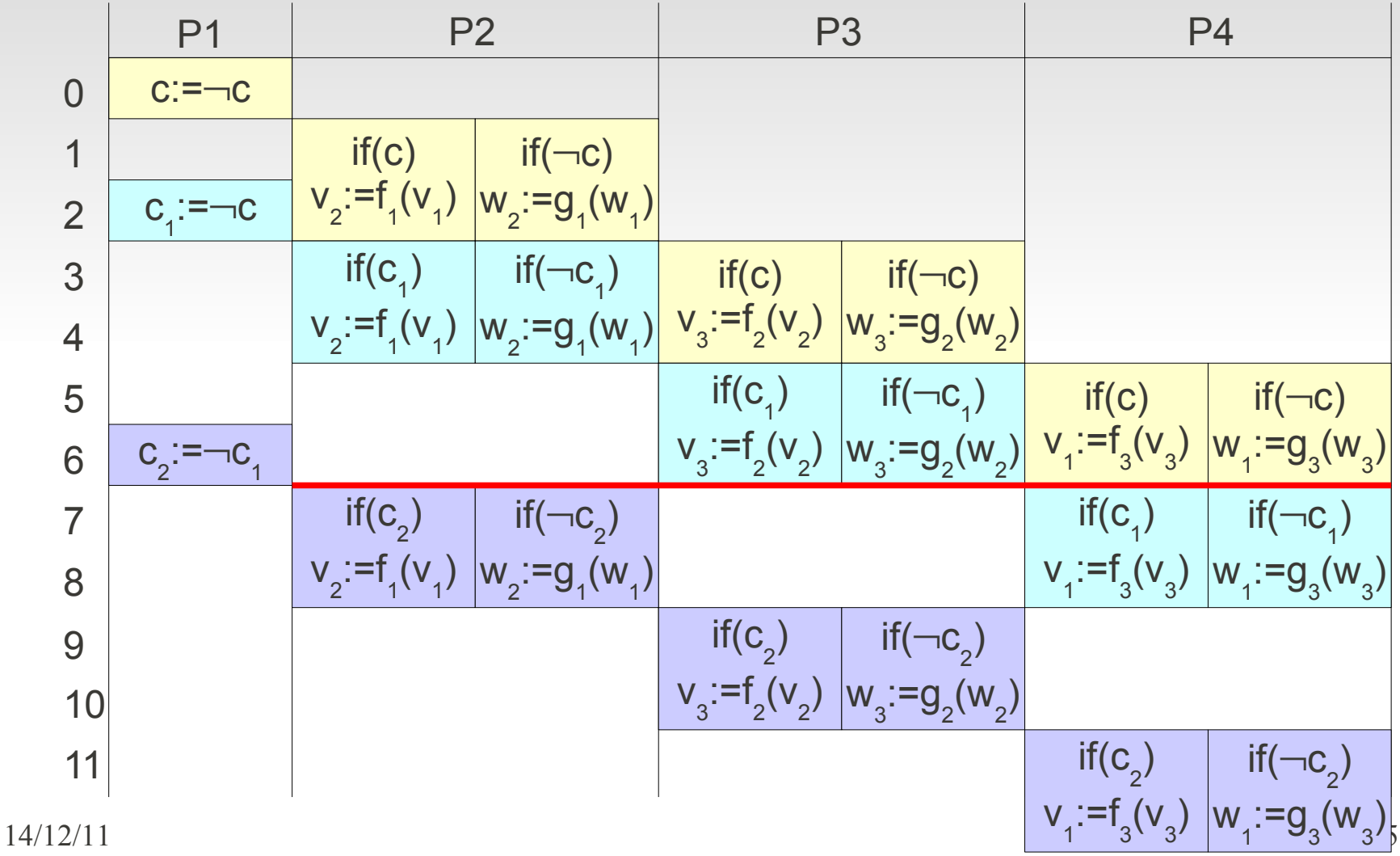
Pipelining algorithm



Pipelining algorithm

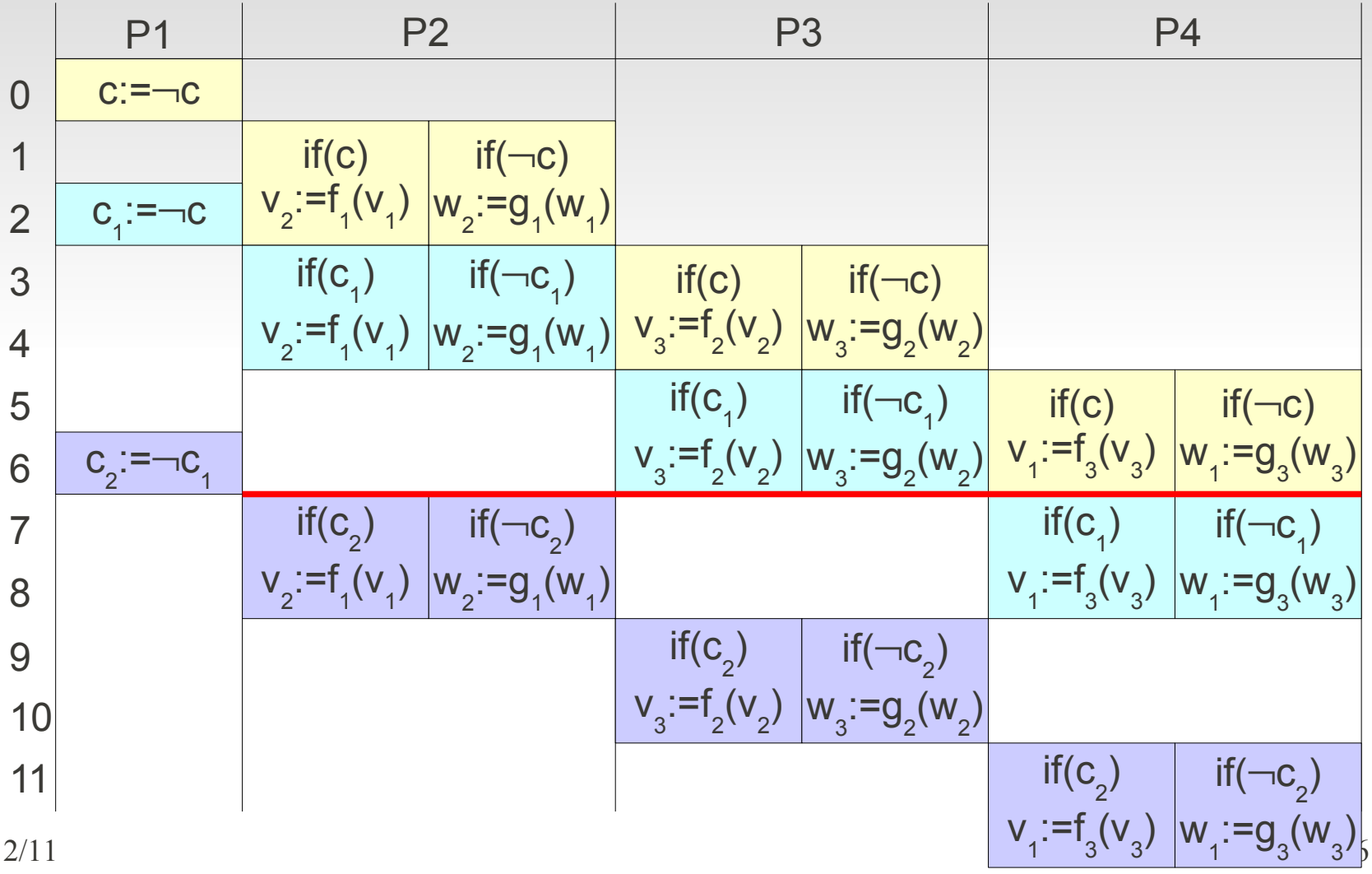


Pipelining algorithm



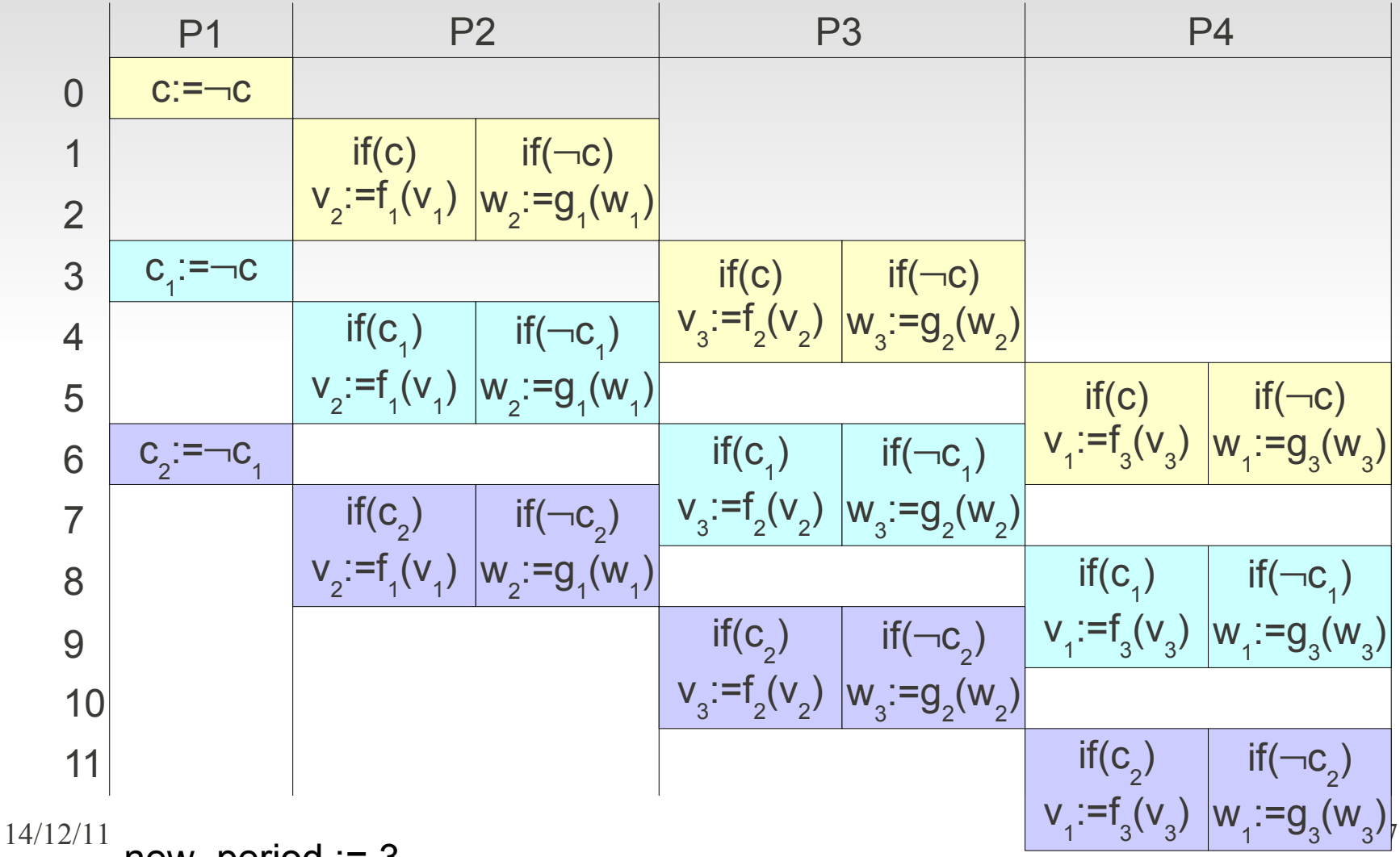
Complete algorithm: the first repetition is fully covered

Pipelining algorithm



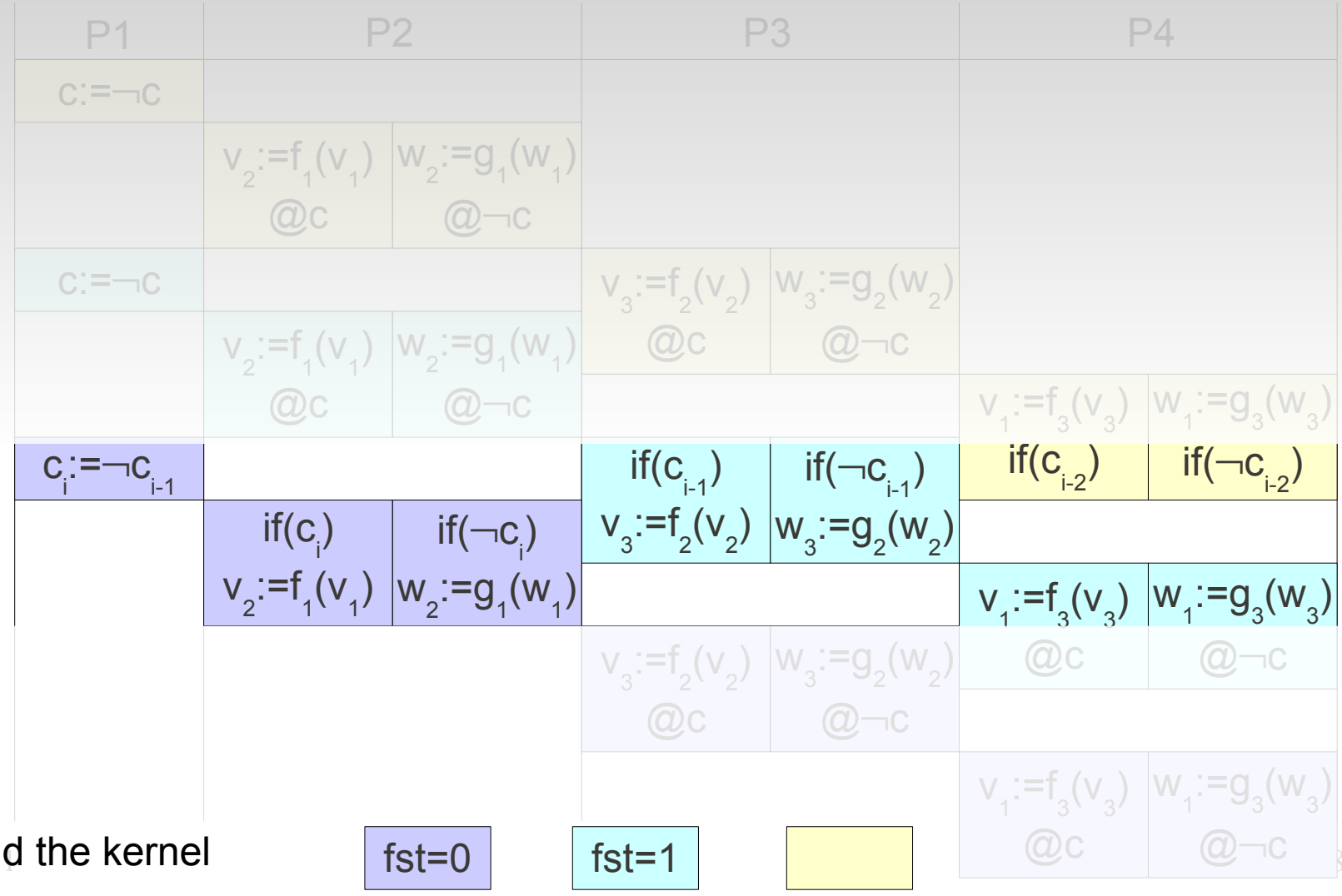
Make the repetition periodic : new_period = $\max_{(o_1,o_2,n)\in DDG}((t(o_1)+d(o_1)-t(o_2))/n)$

Pipelining algorithm



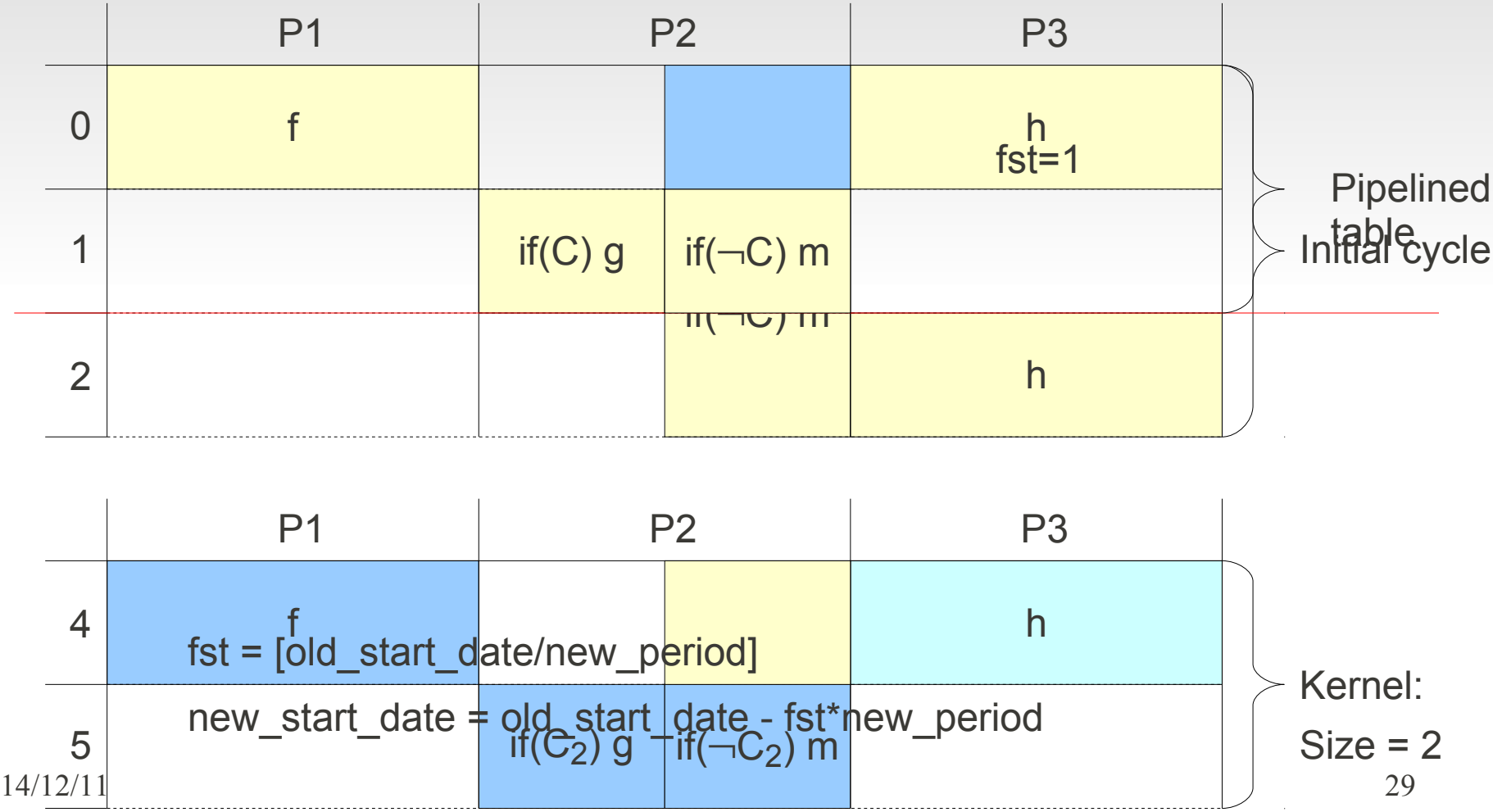
new_period := 3

Pipelining algorithm

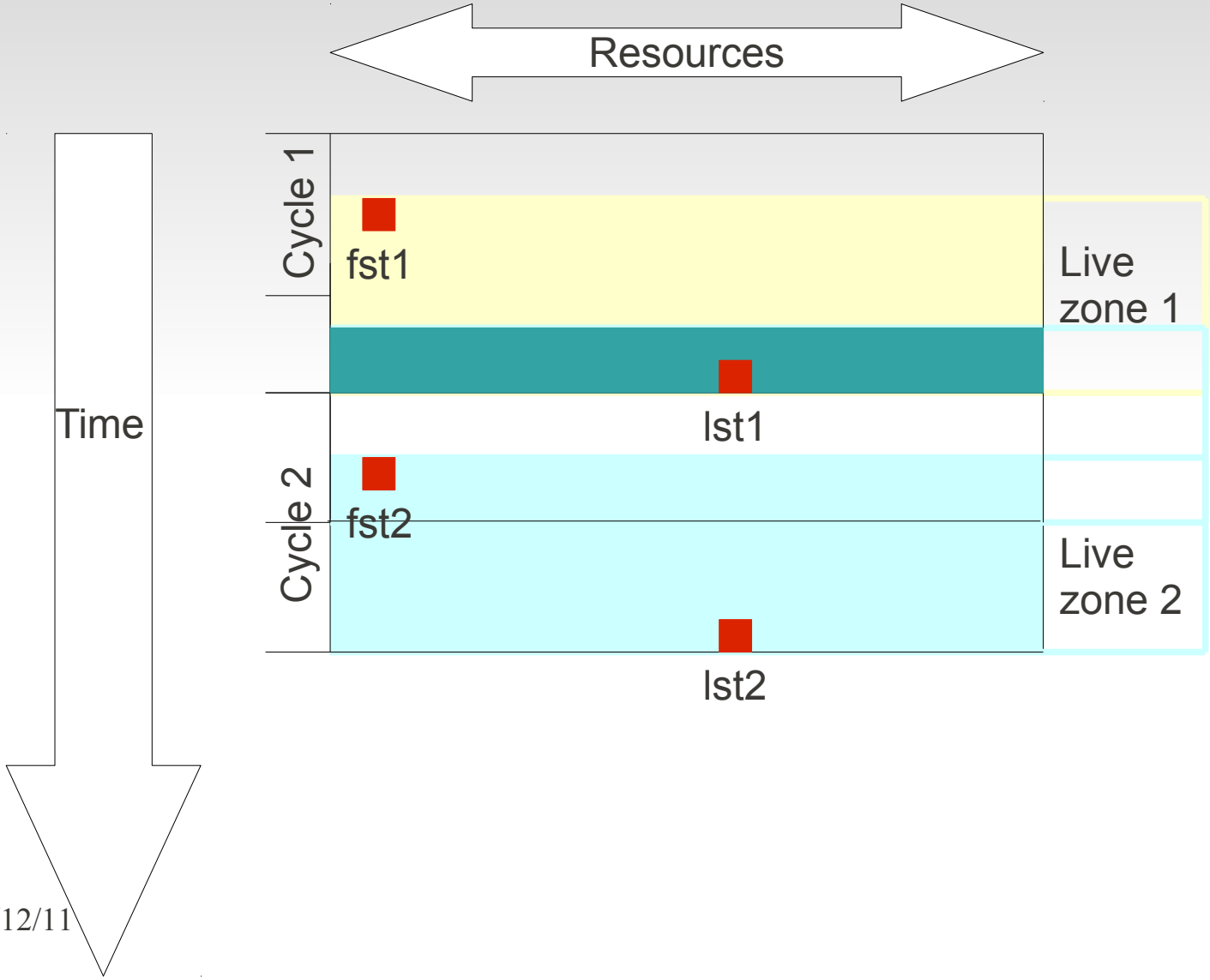


Build the kernel

Pipelined Reservation/Scheduling table construction



Memory aspects



Memory aspects

Problem: if a variable is used at the same time in two or more pipelined iterations, that variable must be **replicated**.

Memory management is achieved the following way:

- The replication factor rep(v) is computed for each variable v,
- Each memory cell v of the initial non-pipelined scheduling table is replaced by **rep(v) memory cells,** allocated on the same memory block as v, in a **cyclic fashion to the successive computation cycles**.

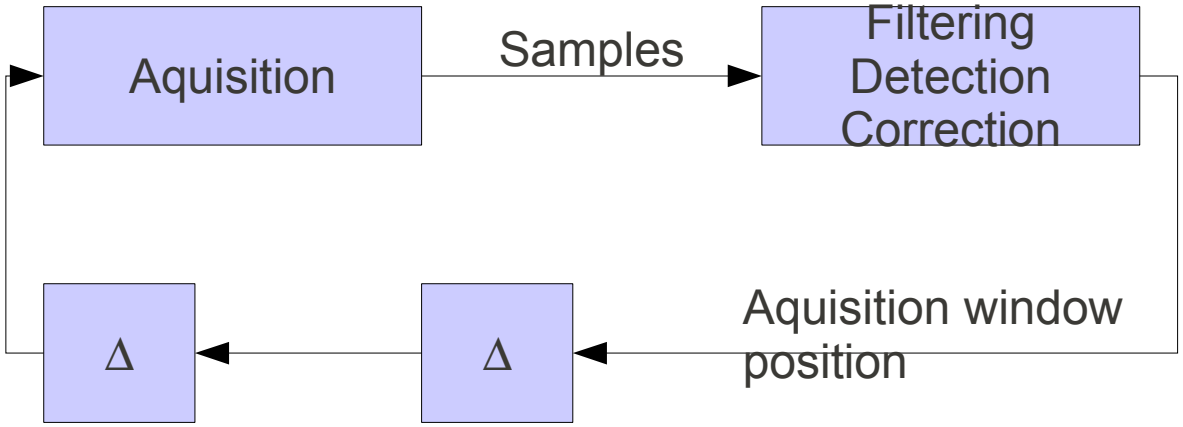
Knock controller example

Industrial case study

Function of the engine control unit for gasoline spark-ignition engines

Objective : choose ignition time for each cylinder at each rotation in order to maximize power output and avoid autoignition as much as possible

High-level description of the control loop function :

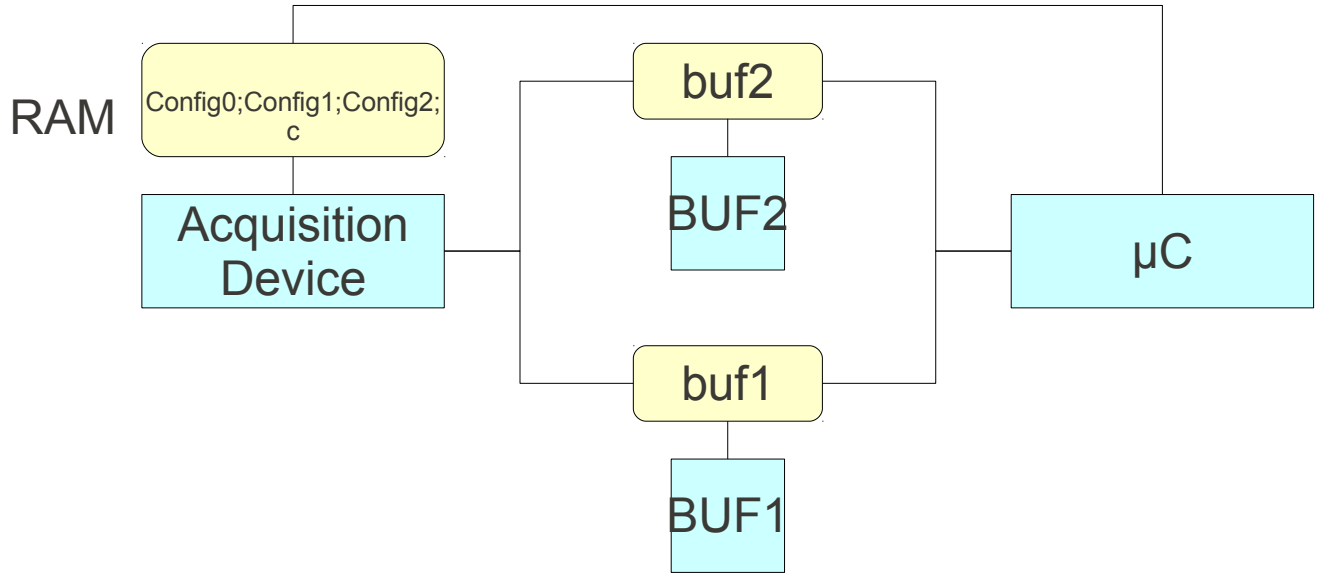


Knock controller example

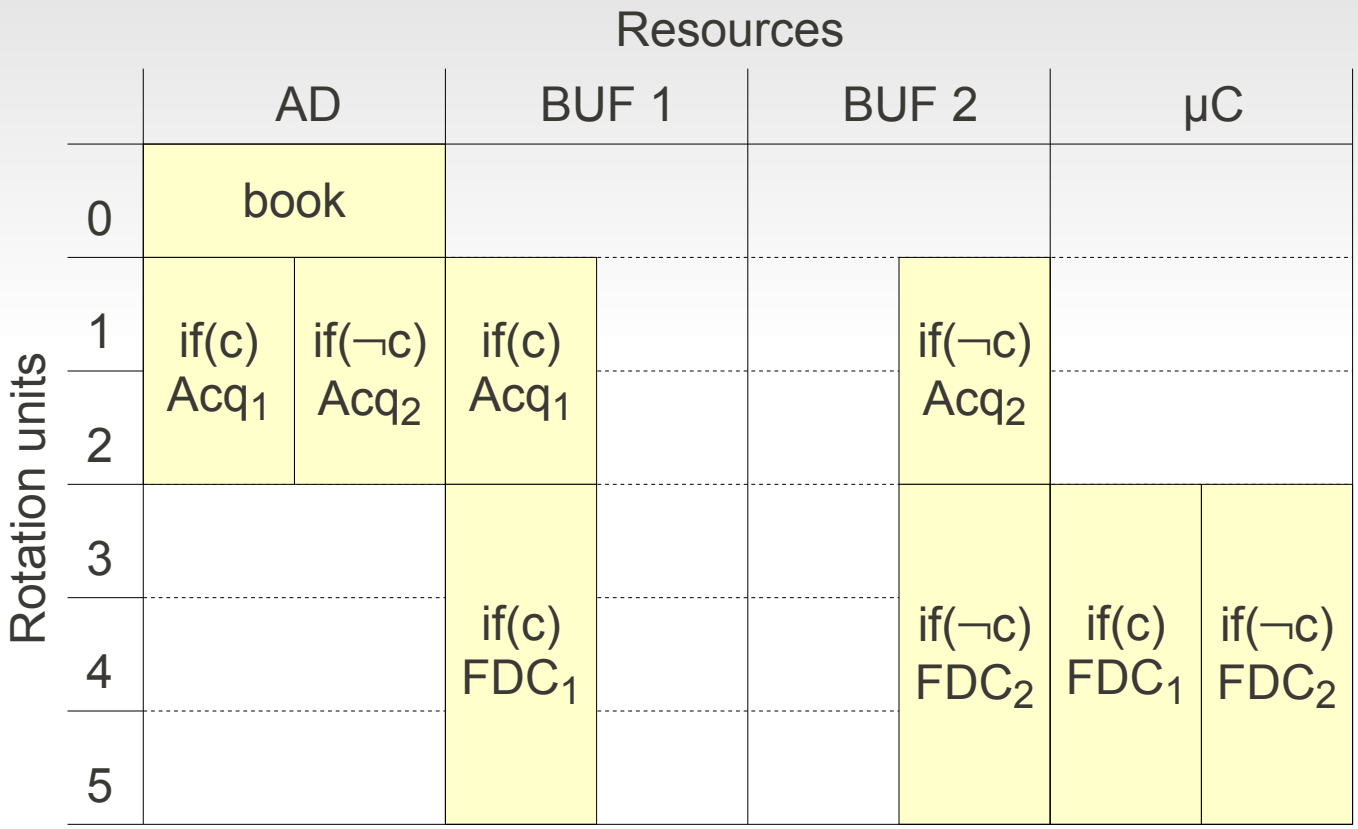
One acquisition device

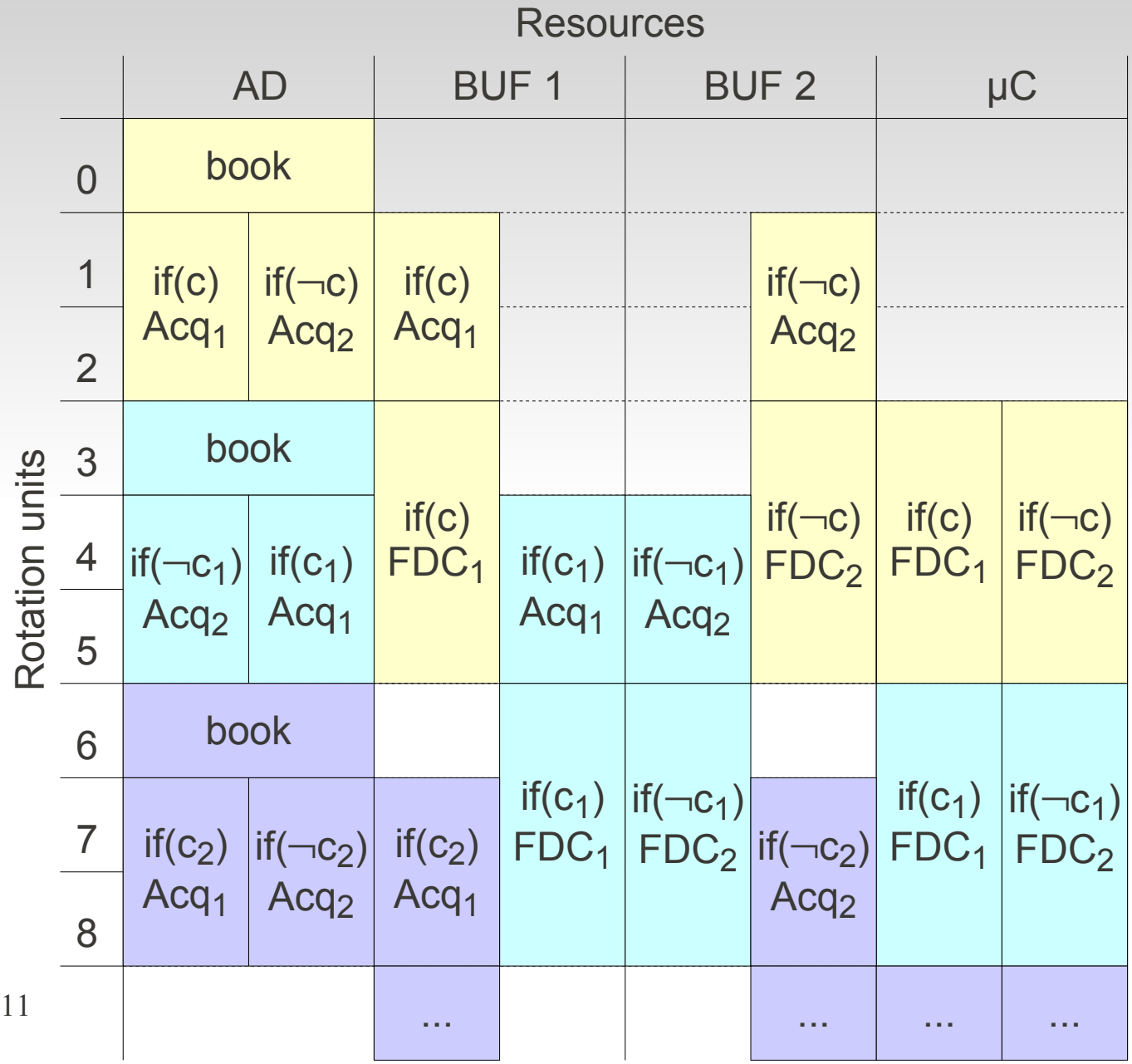
One microcontroller for filtering, detection, and correction

Two buffers and their independent controllers

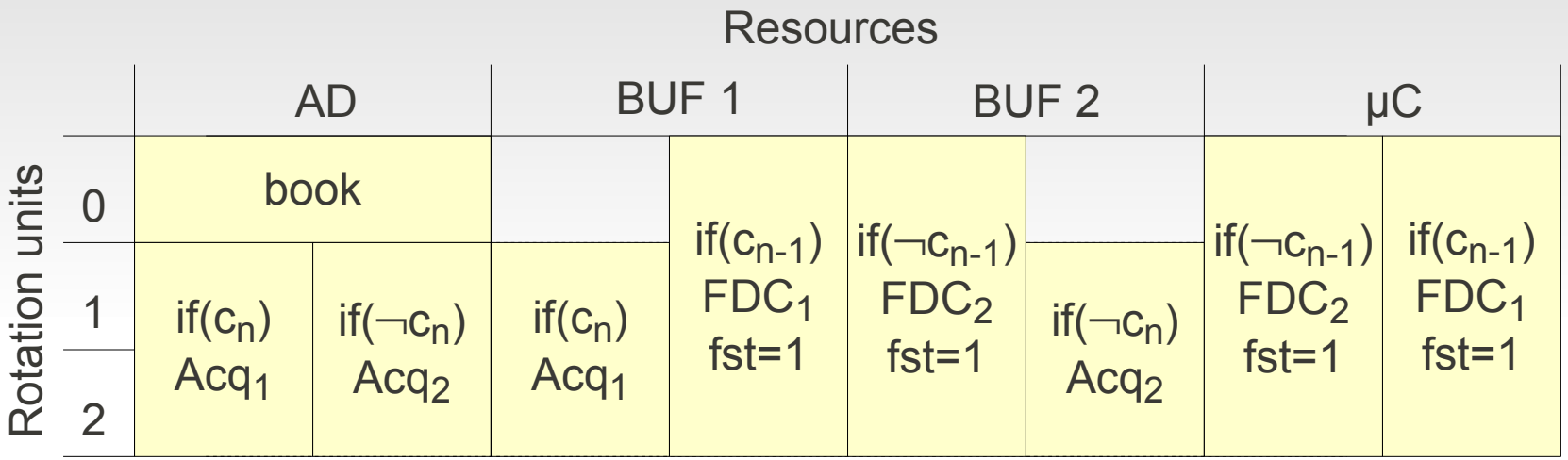


Knock controller example




Resources
AD
BUF 1
BUF 2
µC
Rotation units
0
1
2
3
4
5
6
7
8
book
if(c) Acq1
if(¬c) Acq2
if(c) Acq1
if(¬c) Acq2
book
if(¬c1) Acq2
if(c1) Acq1
if(c) FDC1
if(c1) Acq1
if(¬c1) Acq2
if(¬c) FDC2
if(c) FDC1
if(¬c) FDC2
book
if(c2) Acq1
if(¬c2) Acq2
if(c2) Acq1
if(c1) FDC1
if(¬c1) FDC2
if(¬c2) Acq2
if(c1) FDC1
if(¬c1) FDC2
...
...
...
...

Knock controller example



Experimental results

We demonstrated our algorithms on four examples:

- Embedded control application for the CyCab electric car: 27% reduction in cycle time
- Adaptive equalizer: 6 % reduction
- Knock controller: 50% reduction
- Simple example: 66% reduction. 100% resource usage

Conclusion

- Study of the state of the art in software pipelining
- Definition of formal models
- Definition of algorithms
- Implementation in a prototype
- Evaluation on study cases

Future work

- Enhance memory management: we can forbid memory replication when it is not necessary, but we cannot yet limit it by giving memory sizes,

- Exploit execution guards over partitioned architectures, maybe by using n-synchronous formalism to express and exploit repetition patterns during the pipelining process,

- Integrate pipelining in the initial scheduling process to obtain better trade-offs between latency/response-time, throughput, and resource usage: we would like to do this on a software radio use case we are currently working on.