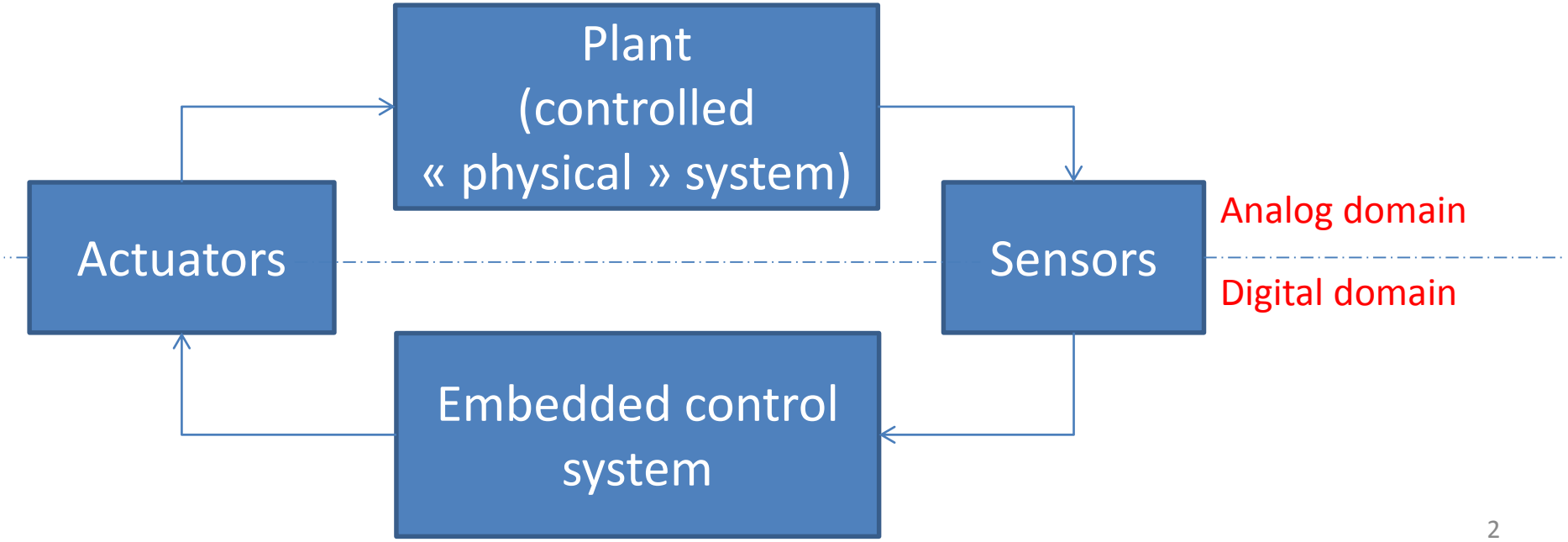
Real Time Systems Compilation with Lopht

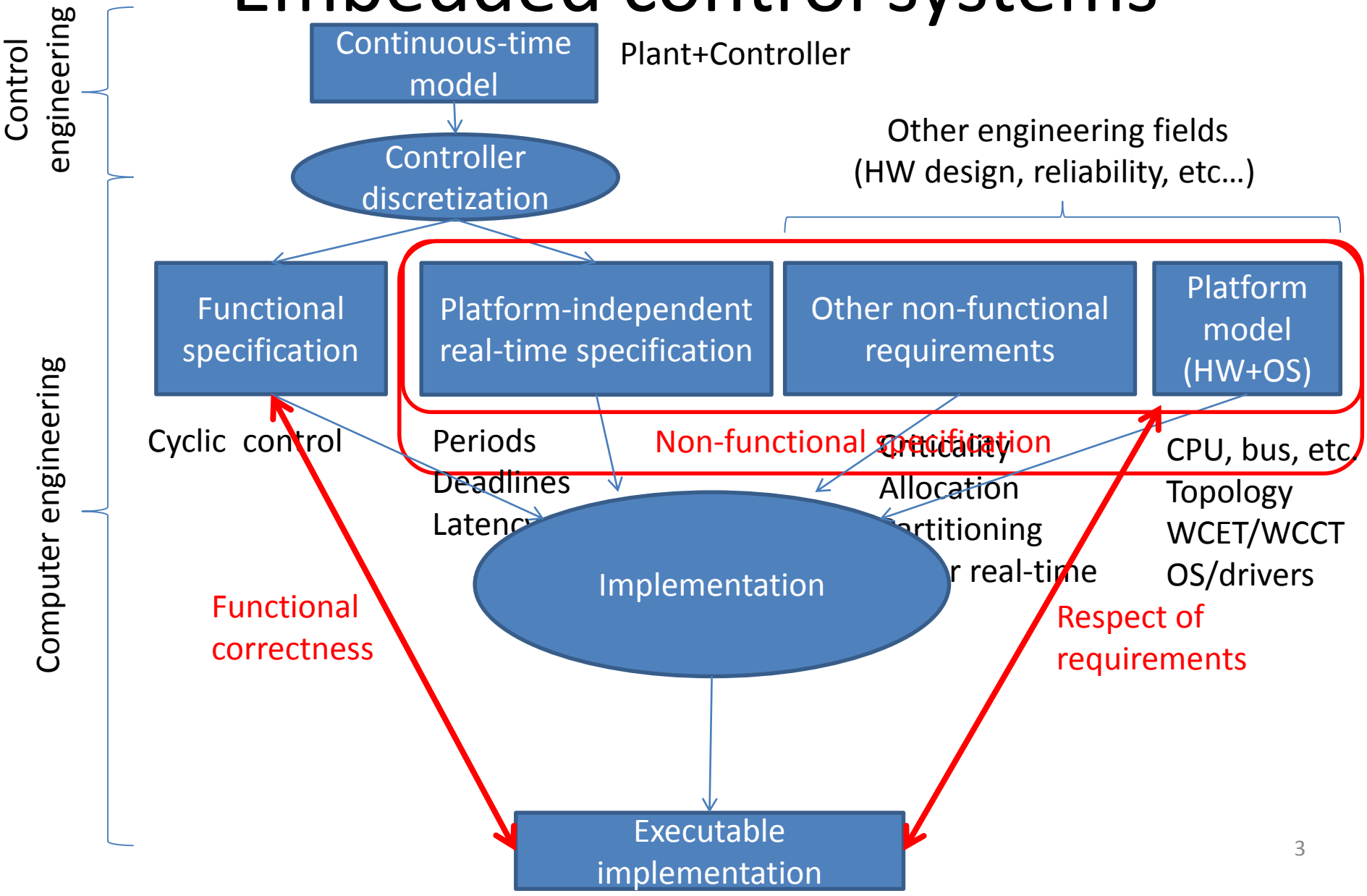
Dumitru Potop-Butucaru
INRIA Paris/ AOSTE team
Jan. 2016

Embedded control systems

- Computing system that controls the execution of a piece of « physical » equipment, in order to ensure its correct functioning



Embedded control systems



Embedded systems implementation

- Still largely a craft in the industry
  - Important manual and/or unformalized phases
    - Some that could be automatized with existing industry-grade tools (modulo certification arguments)
      - Discretization of the continuous-time model
      - Construction of tasks from pieces of C code
      - Allocation and scheduling of tasks for functional correctness
      - Programming and implementing complex timed behaviors
    - Some that cannot (to my knowledge)
      - Building platform and implementation models for precise and safe timing analysis
      - Allocation & scheduling providing hard guarantees of respect for real-time and other non-functional requirements
  - Resource-consuming, error-prone
  - Ultimately rely on test to check system correctness

Hand-crafting implementations no longer scales

- Complexity explodes
  - Large numbers of resources (multi-/many-cores)
  - Large amount of SW (more tasks, more system SW)
  - More complex hardware, OS, SW
    - Communication networks (NoCs, AFDX, etc.), hierarchical schedulers, dependent tasks
- Efficiency becomes critical
  - Parallelized filters = dependent task systems
    - Can rapidly lose performance if multi-processor scheduling is not efficient
  - Performance highly depends on mapping, need to fine tune all system parameters
    - Allocation, scheduling
    - Memory allocations
    - Synchronizations, communications…
- Delegating mapping to HW/OS results in impredictability
  - Cache coherency, GPUs, dynamic allocation and scheduling, etc.

Off-line automation is needed

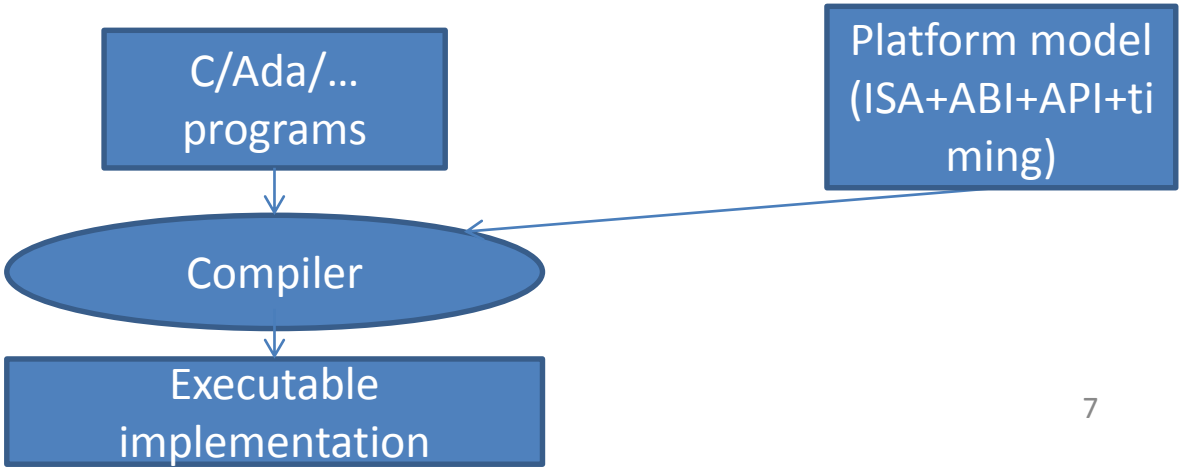
- Requires full formalization:
  - Functional specification
  - Non-functional requirements
  - Execution platform models
    - Sound abstraction issue
  - Formalization of implementation correctness
- Requires efficient algorithms for analysis and synthesis

Standardization is a prerequisite

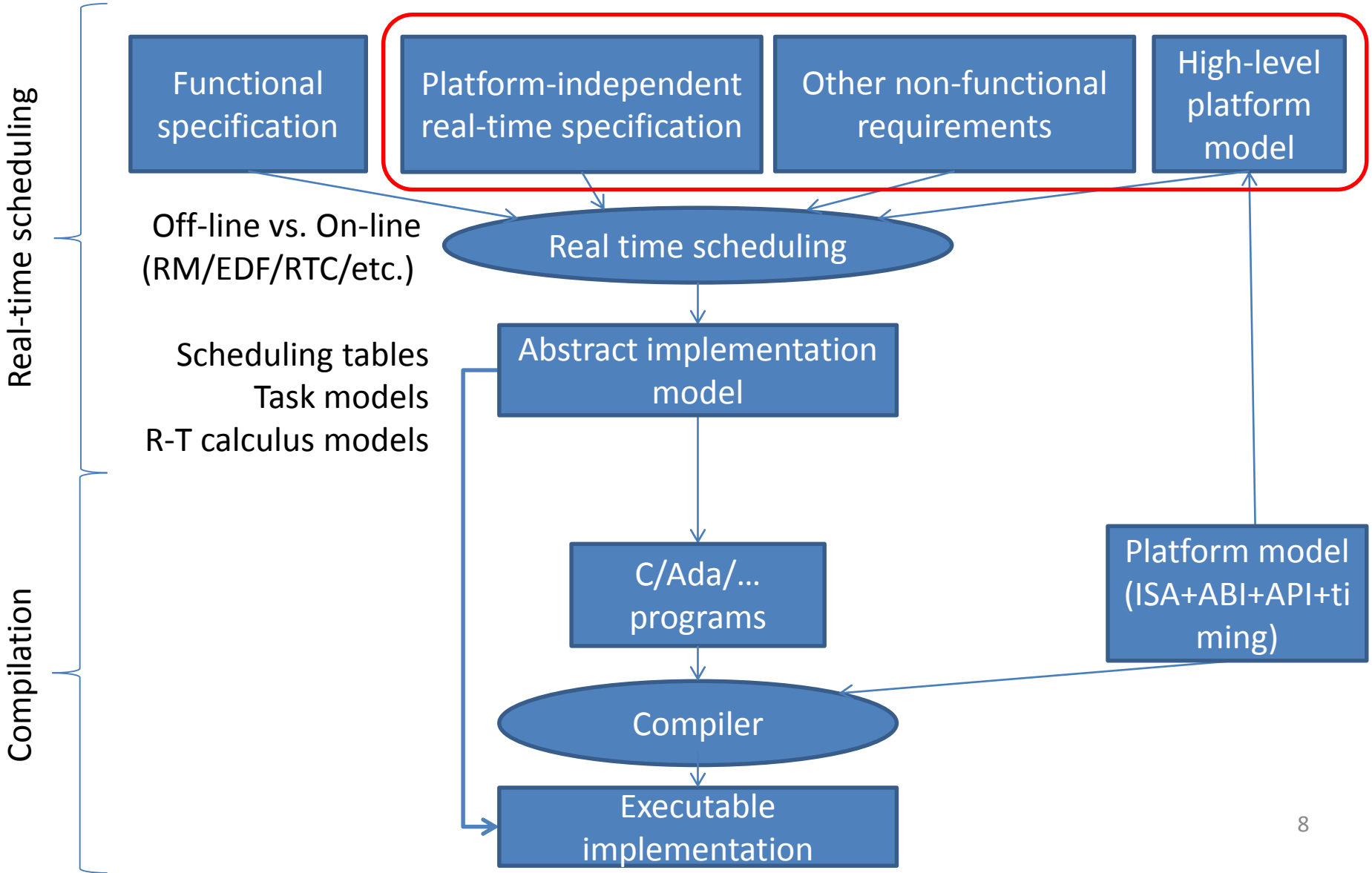
(allows cost-effective tool-building)

A success story of off-line automation: compilation

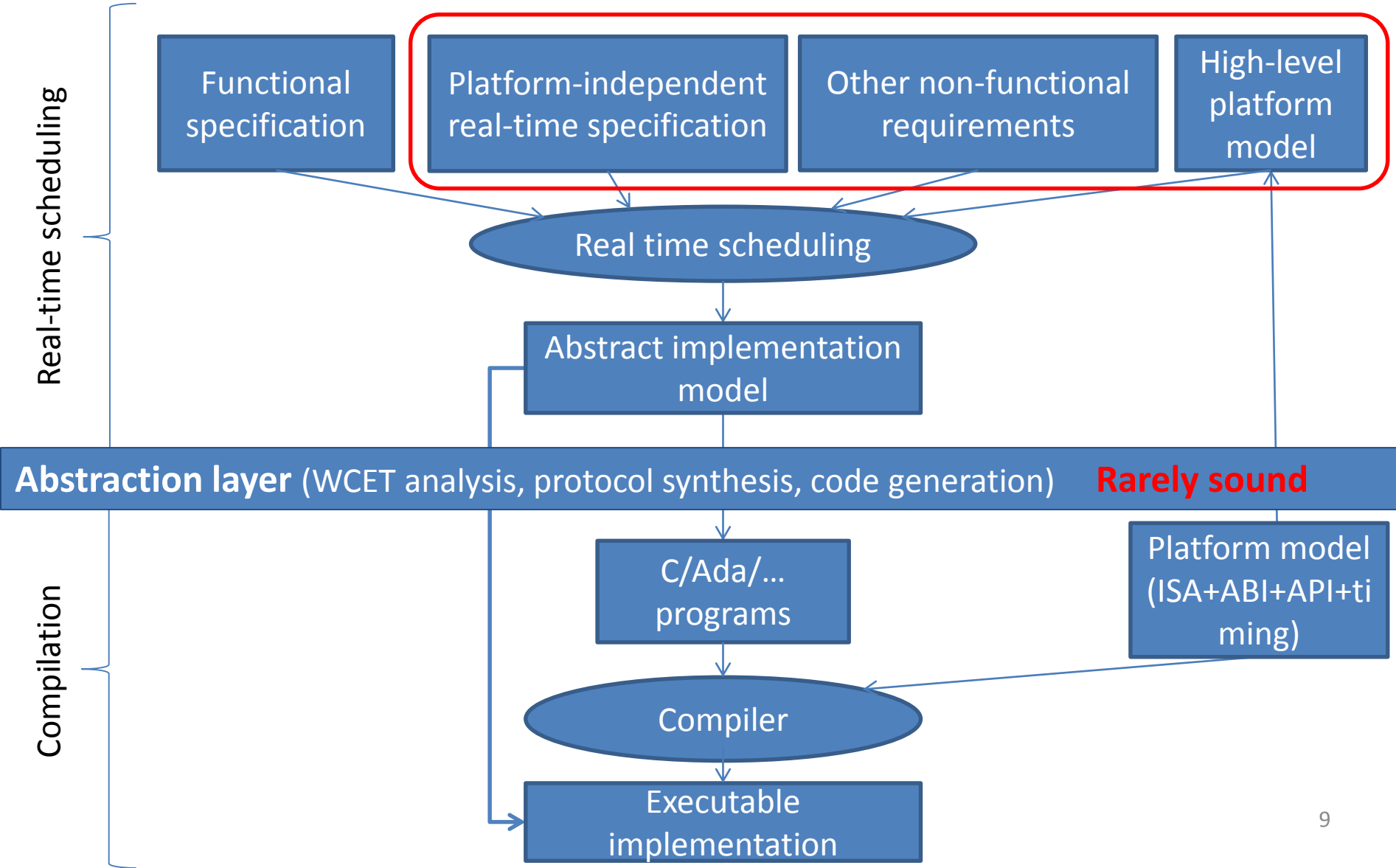
- Low level of the embedded implementation flow.
- Made possible by the early standardization of programming languages, and execution platforms (ISAs, ABIs, APIs).
- Almost complete replacement of assembly coding, even in embedded systems design.



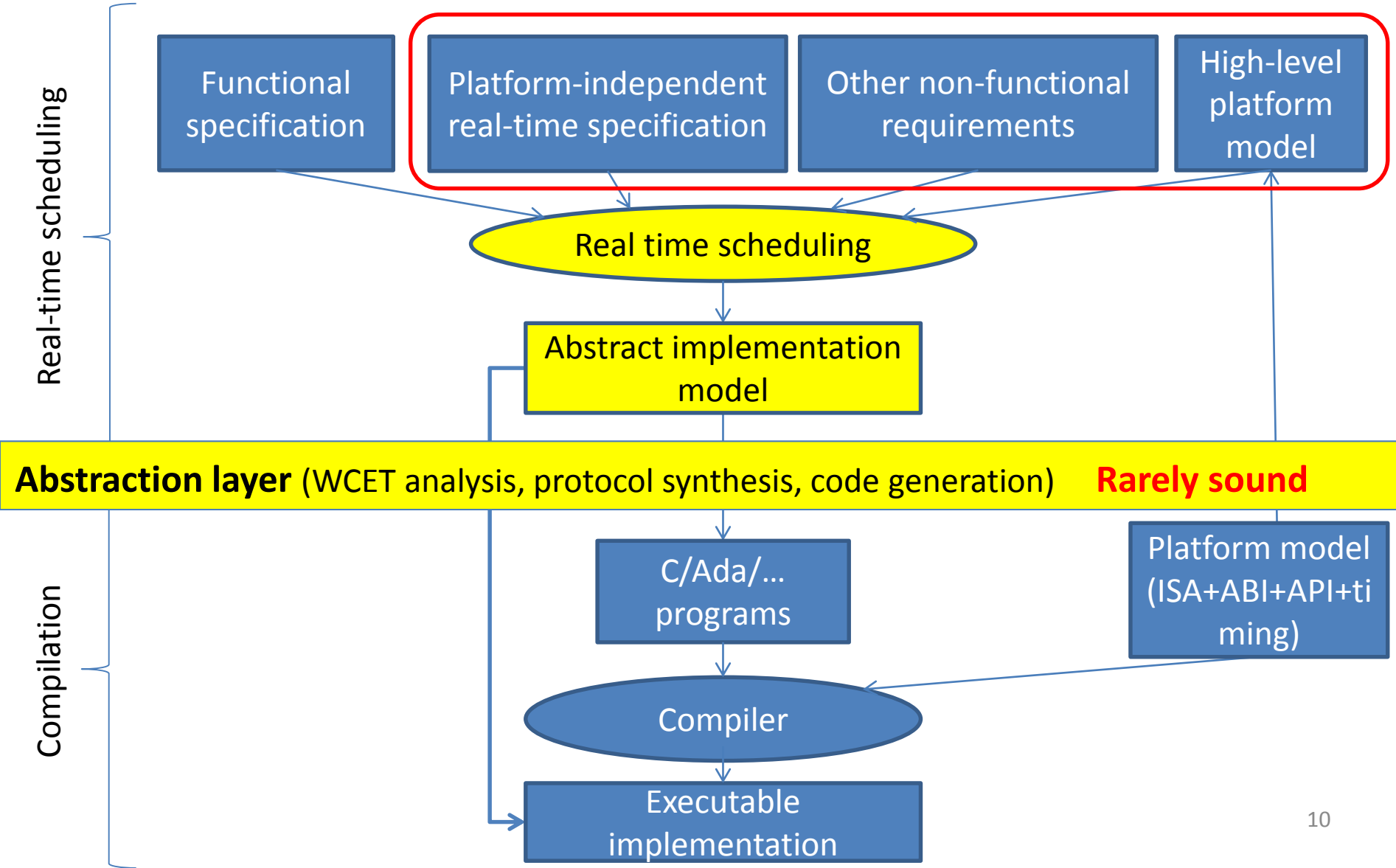
Real-time scheduling vs. compilation



Real-time scheduling vs. compilation



Challenge: full automation



Challenge: full automation

- Historically: difficult, due to lack of standardization
- Recently: functional and non-functional specification languages
  - Dynamic systems specification languages:
    - Simulink, LabView, Scade, etc.
    - Code generation ensuring functional correctness, not schedulability
    - Discretization
  - Systems engineering languages:
    - SysML, AADL…
    - Real-time, Criticality/partitioning, Allocation…
- Today: Execution platforms with support for real-time:
  - ARINC 653, AUTOSAR, TTP/TTEthernet, etc.
  - Many-cores (some of them)

Real time systems compilation

- Move from specification to running implementation
  - Fully automatically
    - No human intervention during the compilation itself
  - Inputs and outputs must all be formalized
    - Functional and non-functional inputs, (model of) generated code
    - Provide formal correctness guarantees (functional&non-functional)
    - No break in the formal chain
      - Formal proof possible
  - Failure traceability
    - Functional and non-functional causes
    - Easy to do trial-and-error systems design
  - **Fast and efficient**
    - **Fine-grain platform and application modeling**
    - **Use of fast heuristics**
      - **Compilation, real time scheduling, synchronous languages, etc.**
      - **Exact techniques (e.g. SMT-based) do not scale [FORMATS'15]**

Real time systems compilation

- Start with statically scheduled systems
  - Timing analysis-friendly (if HW&OS are also friendly)
    - Good performance and timing predictability
    - Checking functional correctness and computing WCET/WCRT is easy using existing tools
    - We have good experience with static scheduling approaches
  - Classes of applications with practical importance
    - Critical embedded control applications
    - Signal/image processing
  - Implementation model: scheduling/reservation tables
    - Shared between real-time scheduling and compilation
      - » Facilitate reuse of algorithms

Previous work

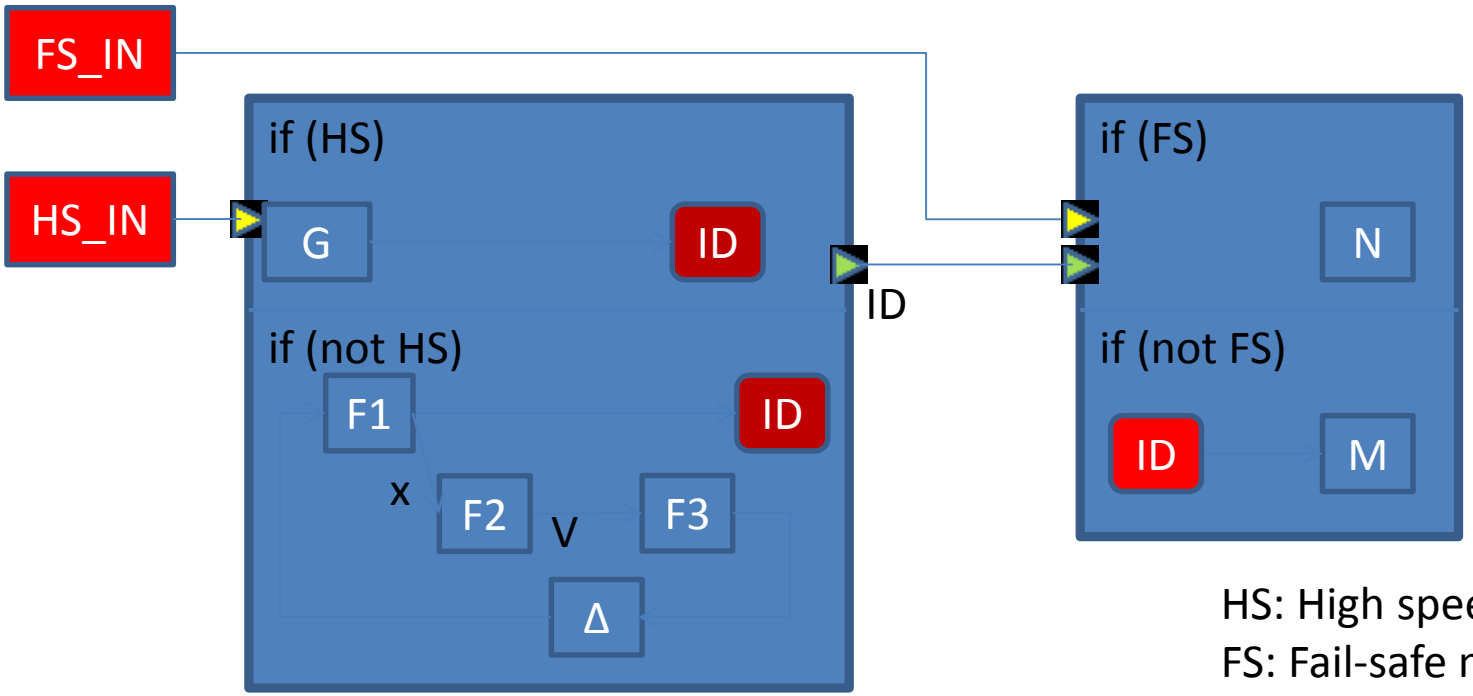
- Work on superscalar/VLIW/many-core compilation
  - Optimization, not respect of requirements
  - No hard real-time guarantees
  - Finer level of control
- Work on off-line real-time scheduling
  - AAA/SynDEx (Sorel et al.)
    - Does not have: classical compiler optimizations (pipelining), fine-grain execution condition analysis, time triggered code generation, partitioning, preemptable tasks, memory allocation…
  - Simulink -> Scade -> TTA (Caspi et al.)
    - Does not have: Automatic allocation, conditional execution, preemption, mapping optimizations (pipelining, memory bank allocation, etc.).
  - Prelude (Forget et al.)
    - Does not have: Partitioning, deadlines longer than periods, memory bank allocations.
  - PsyC/PharOS
    - Input is lower-level (can be used as back-end to our approach)

The real time compiler Lopht

- Compiler for statically-scheduled real-time systems
  - Functional specification: Data-flow synchronous (e.g. Lustre/Scade)
  - Target execution platforms:
    - Distributed time-triggered systems
      - ARINC 653-based processors, TTEthernet
      - Full generation of ARINC 653 configuration & partition C/APEX code & bus schedule
    - « WCET-friendly » many-cores (our own, Kalray MPPA)
      - Full generation of C bare metal code, including communication and synchronization operations, memory allocations, NoC configuration, etc.
      - WCET analysis of generated parallel code, including synchronization code (sound architecture abstraction layer)
  - Meaningful applications:
    - Space launcher model (Airbus DS)
      - Complex non-functional requirements
    - Passenger exchange (Alstom/IRT SystemX)
      - Focus on mixed criticality
    - Platooning application model (CyCab)
      - Parallelized image processing code inside a control loop

The principle: table-based scheduling

- Example: Engine ignition application

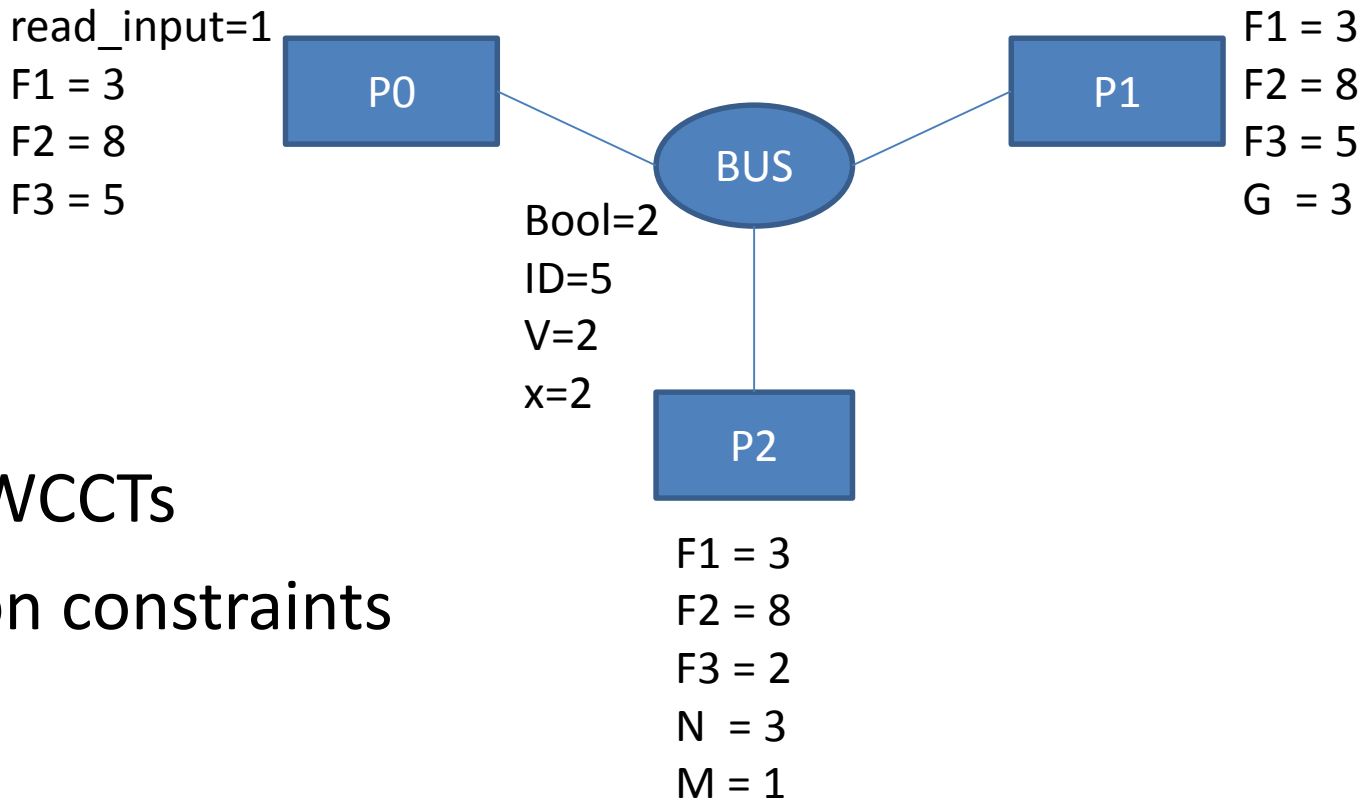


HS: High speed mode
FS: Fail-safe mode

  - Functional specification: dataflow synchronous
    - Cyclic execution model
    - 4 modes determined by 2 independent switches

The principle: table-based scheduling

- Example : Non-functional specification
  - 3 CPUs, 1 broadcast bus



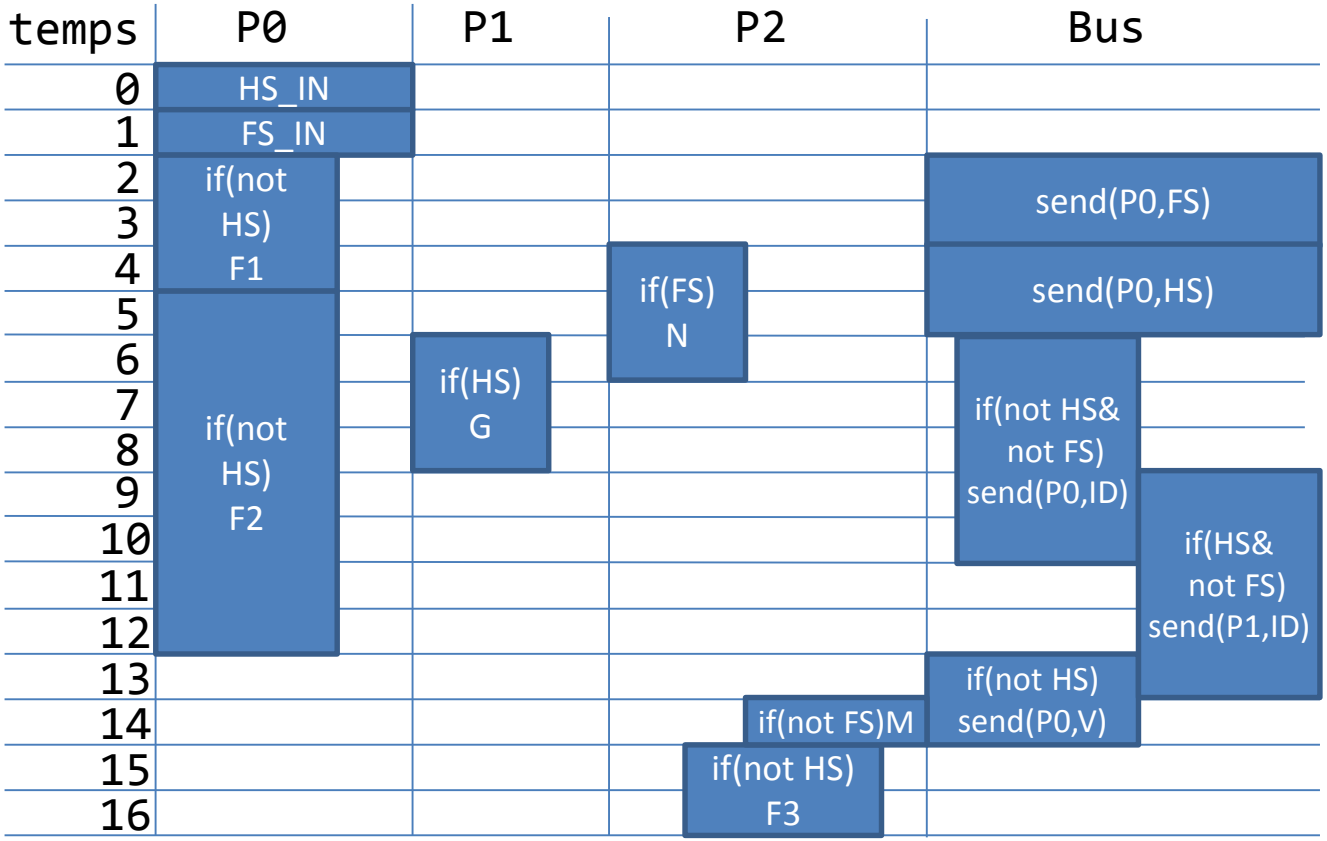
  - WCETs/WCCTs
  - Allocation constraints

The principle: table-based scheduling

- List scheduling (like in compilers, SynDEx, etc.)
  - Allocate and schedule the blocks 1 by 1
  - Allocation and scheduling are optimal for each block
    - Criterion: Cost function
      - All legal allocations are tried, we retain the one that minimizes the cost function
      - Cost functions:
        - » Simplest: worst-case end date of the block (this example)
        - » More complex can include e.g. locality-related terms
    - Does not imply global optimality (heuristic)
      - No backtracking
  - Communication mapping is done during block mapping
  - **Failure reporting: current scheduling status, blocking reason**

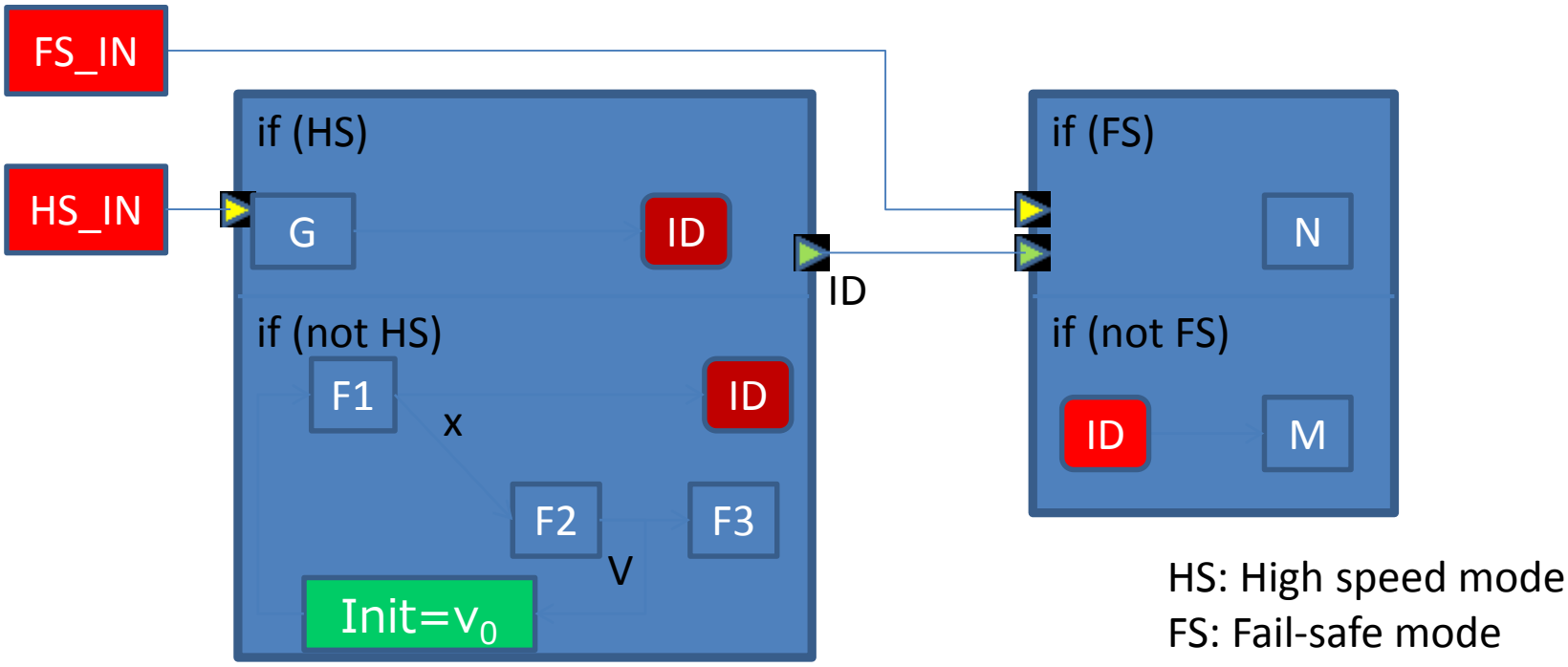
The principle: table-based scheduling

- Result: conditioned scheduling table



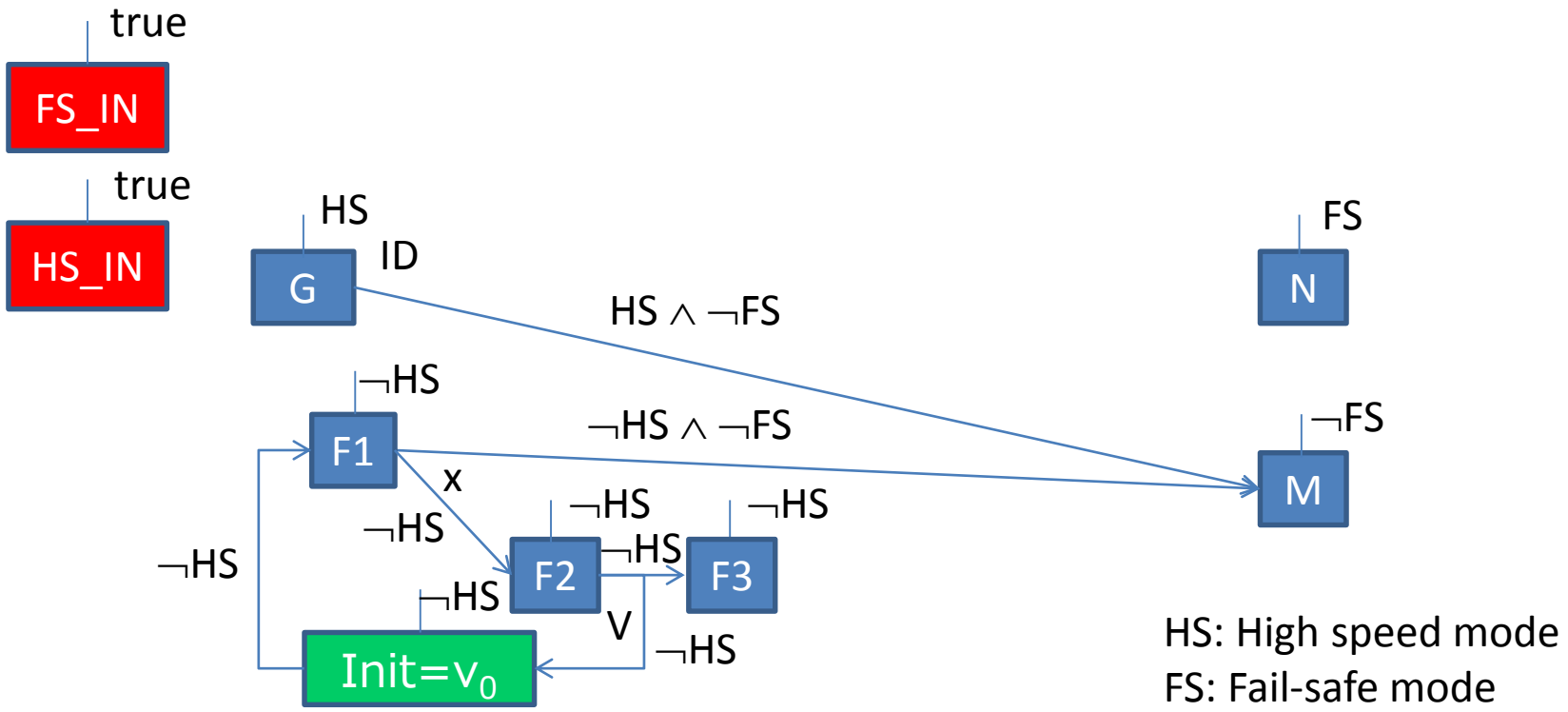
Before scheduling: flattening the dataflow

- Translation to non-hierarchic data-flow



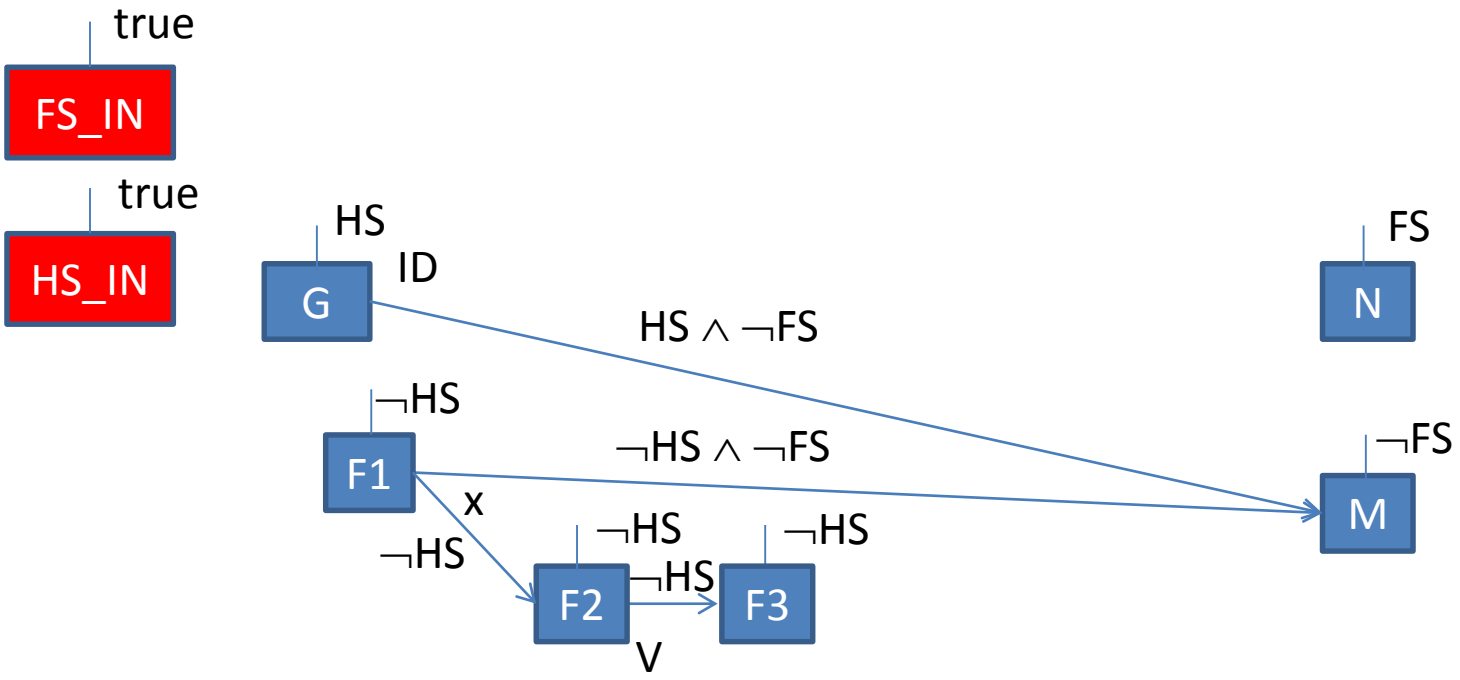
Before scheduling: flattening the dataflow

- Translation to non-hierarchic data-flow



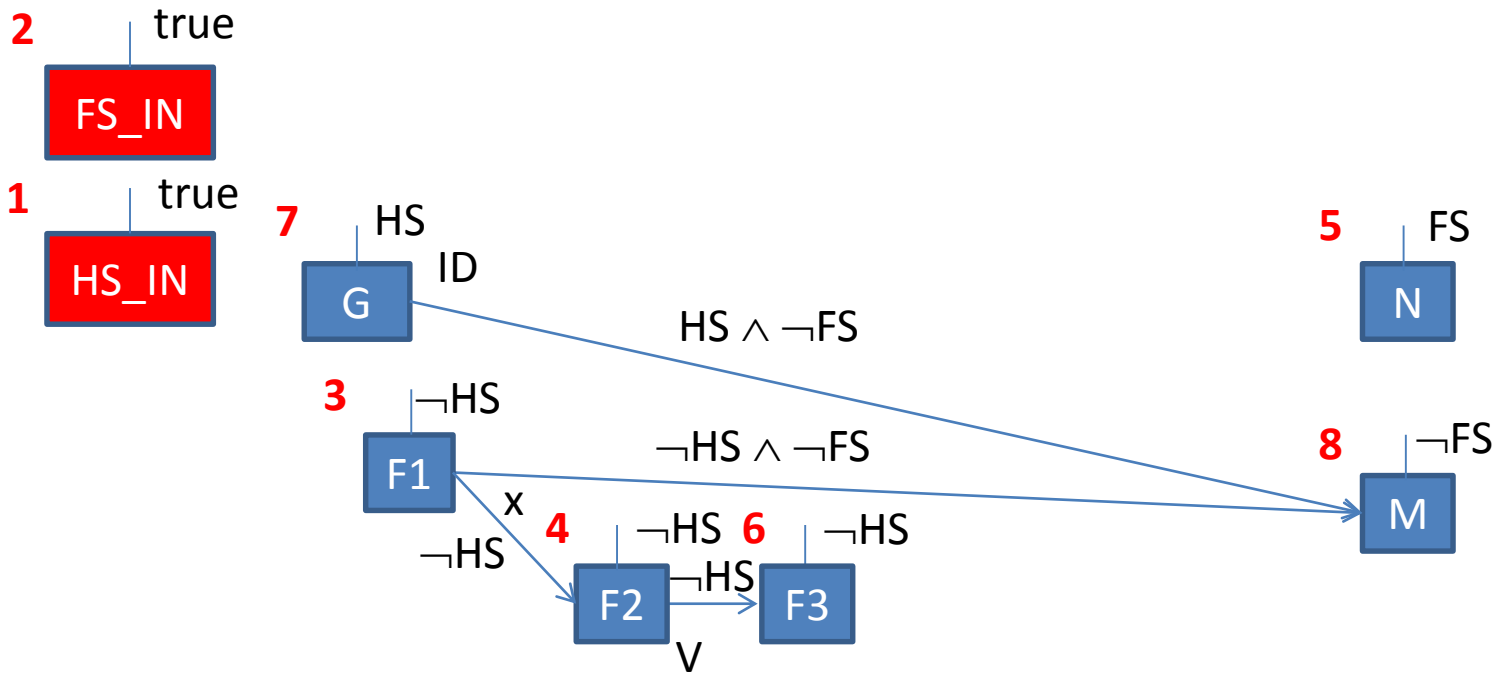
Scheduling algorithm

- Step 1: order the dataflow blocks (« list scheduling »)
  - Total order compatible with the data dependencies (except for delayed ones)



Scheduling algorithm

- Step 1: order the dataflow blocks (« list scheduling »)
  - Total order compatible with the data dependencies (except for delayed ones)



Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1
  - Allocation and scheduling are optimal for each block, taken separately.
    - Criterion: Cost function
      - Try all legal allocations, but retain only one of those that minimize the cost function
      - In this example, cost function = reservation end date
    - Local optimality does not imply global optimality
      - No backtracking
  - Communication scheduling is done on demand, during block scheduling.
    - Multiple routes possible => more complicated

Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1

At first, the scheduling table is empty.

time	P0	P1	P2	Bus
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				

Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1

Operation HS_IN can only be allocated on P0. The optimal schedule places the operation at date 0.

time	P0	P1	P2	Bus
0	HS_IN			
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				

Scheduling algorithm

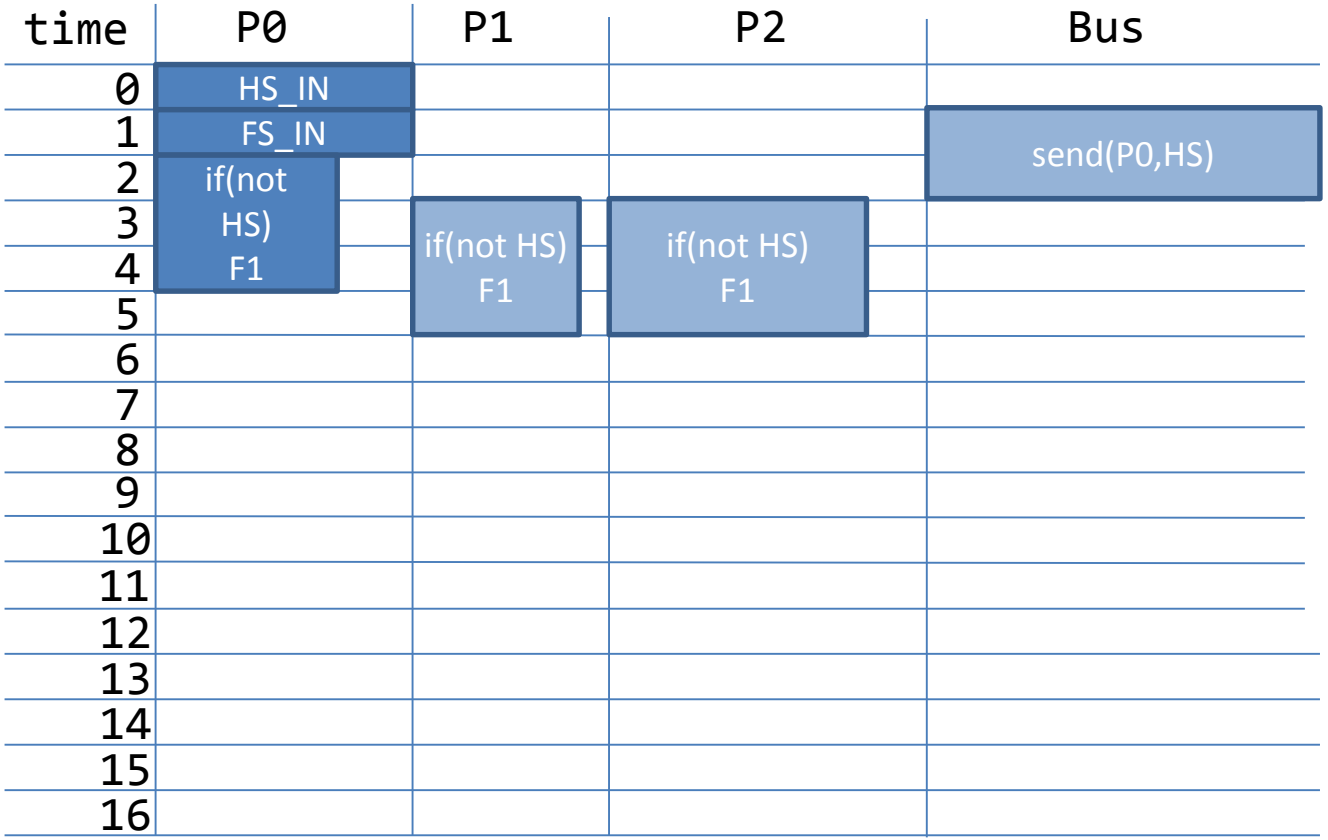
- Step 2: allocate and schedule the blocks 1 by 1

Operation FS_IN can only be allocated on P0. The optimal schedule places the operation at date 1.

time	P0	P1	P2	Bus
0	HS_IN			
1	FS_IN			
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				

Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1



Operation F1 can be allocated on P0, P1, or P2. We retain the allocation on P0 (in darker blue) be-cause in this case F1 terminates at date 5, whereas on P1 or P2 it would end at date 6 (due to the communication of HS, needed to compute the activation condition).

Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1

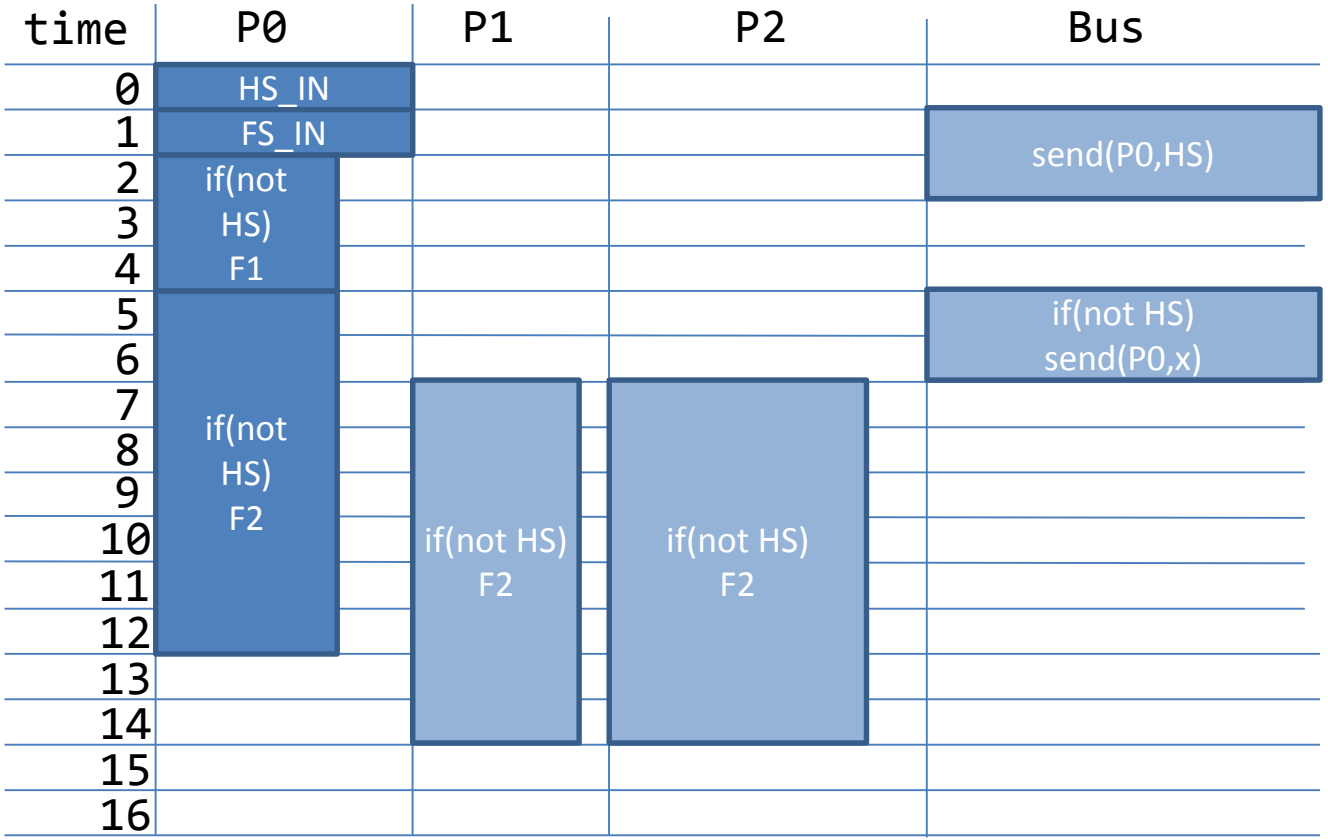
time	P0	P1	P2	Bus
0	HS_IN			
1	FS_IN			
2	if(not			
3	HS)			
4	F1			
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				

Operation F1 can be allocated on P0, P1, or P2. We retain the allocation on P0 (in darker blue) be-cause in this case F1 terminates at date 5, whereas on P1 or P2 it would end at date 6 (due to the communication of HS, needed to compute the activation condition).

Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1

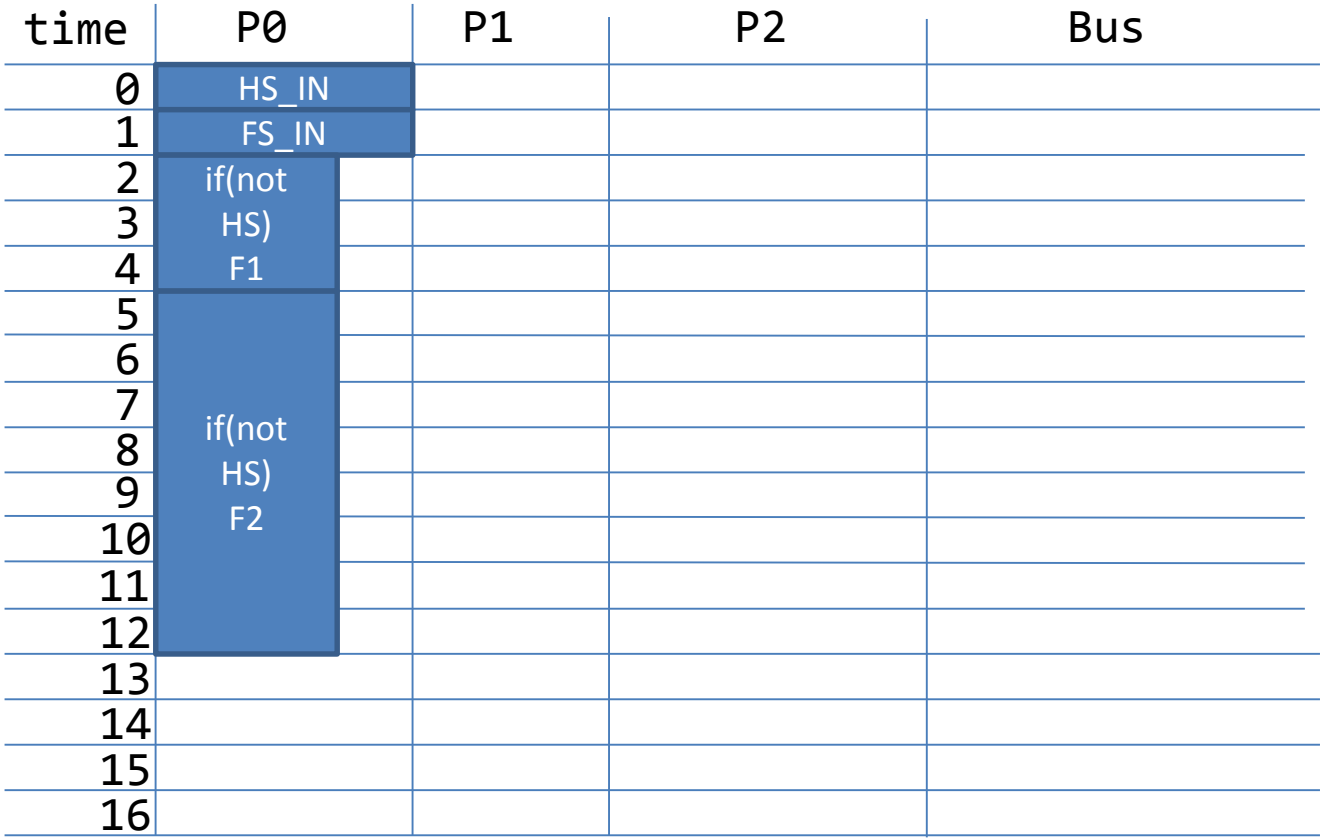
Same for F2.



Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1

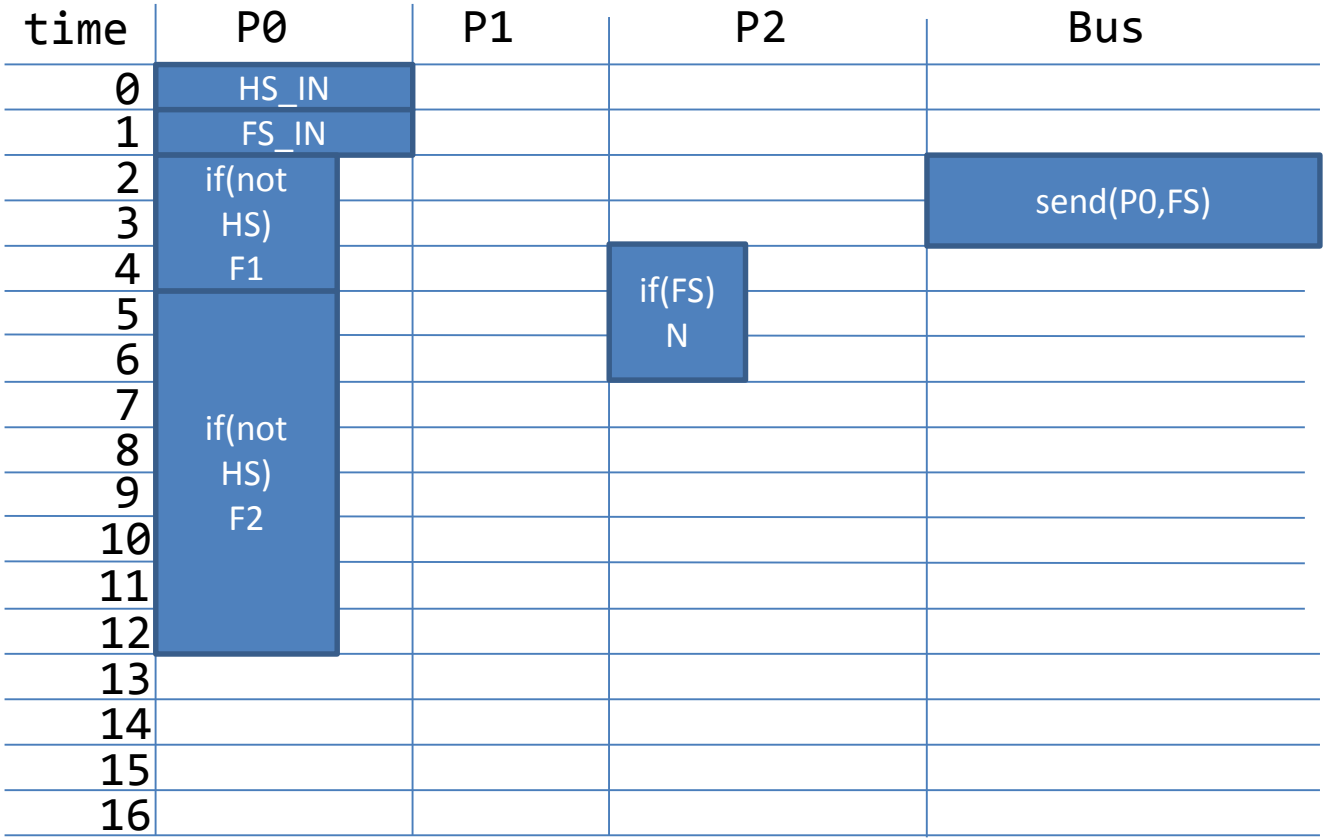
Same for F2.



Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1

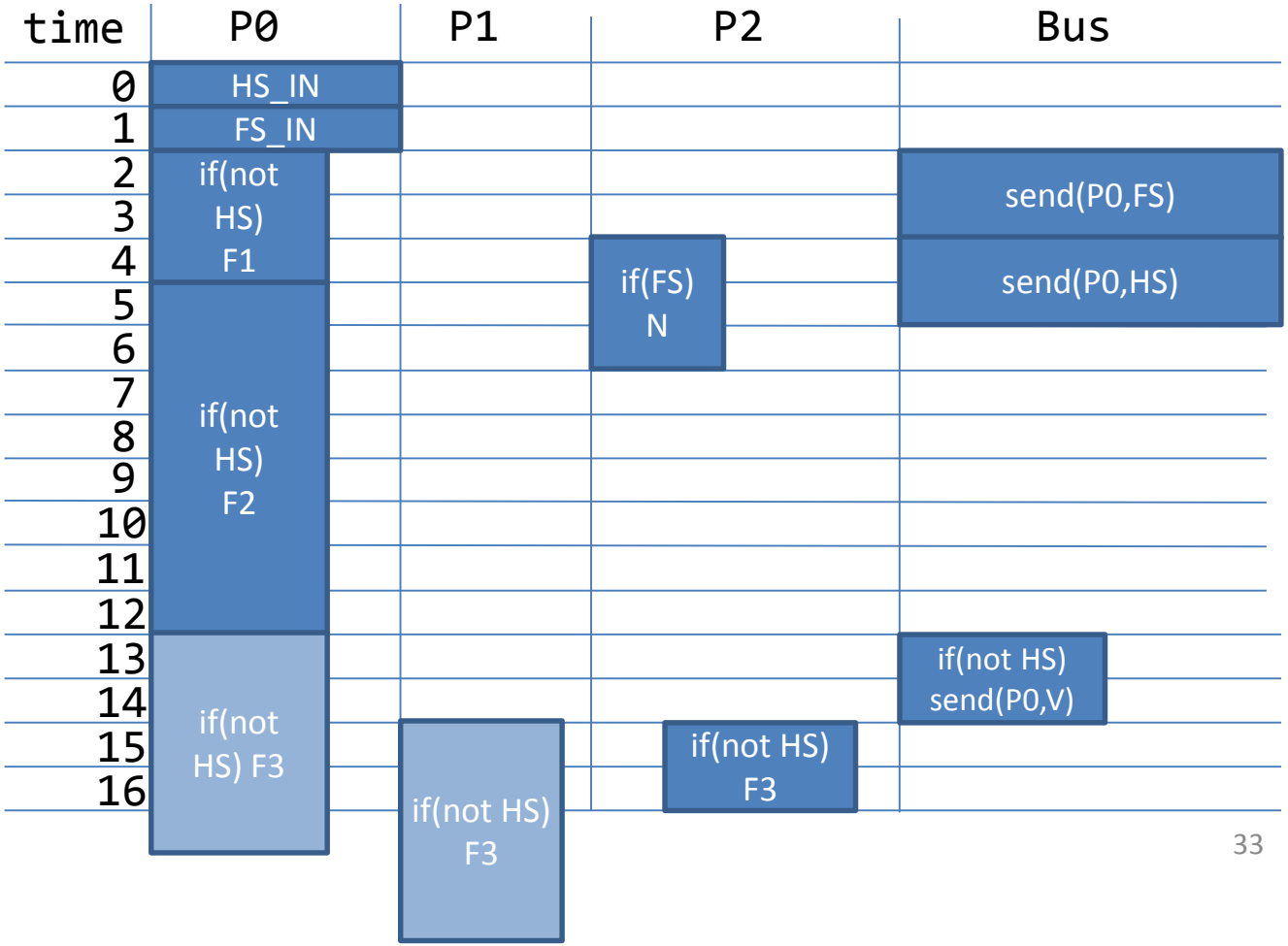
N can only be allocated on P2. This forces the transmission of FS. The 2 operations are scheduled as soon as possible (ASAP).



Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1

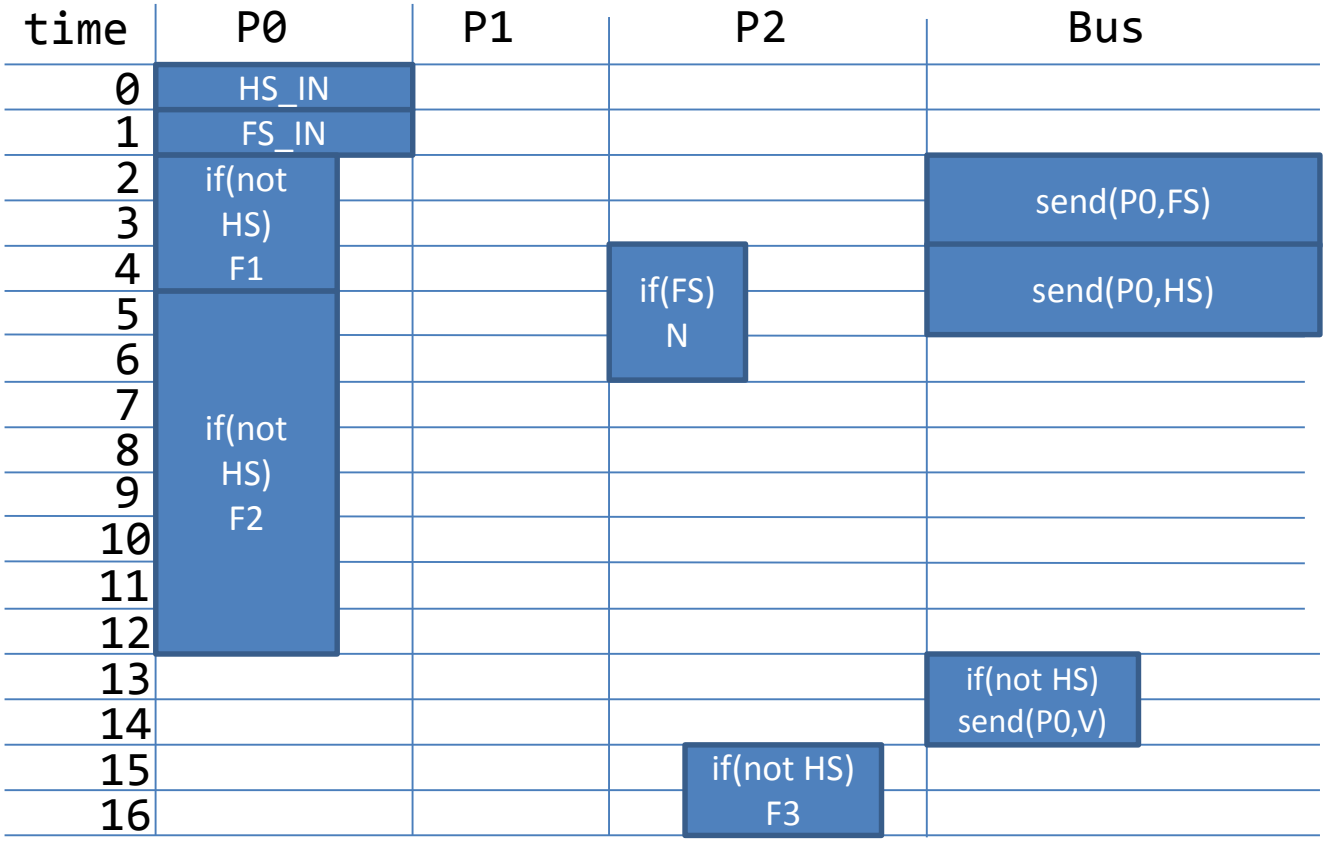
F3 can be allocated on P0, P1, or P2. It is executed much faster on P2, which compensates for the needed communication time. The allocation on P2 is therefore retained.



Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1

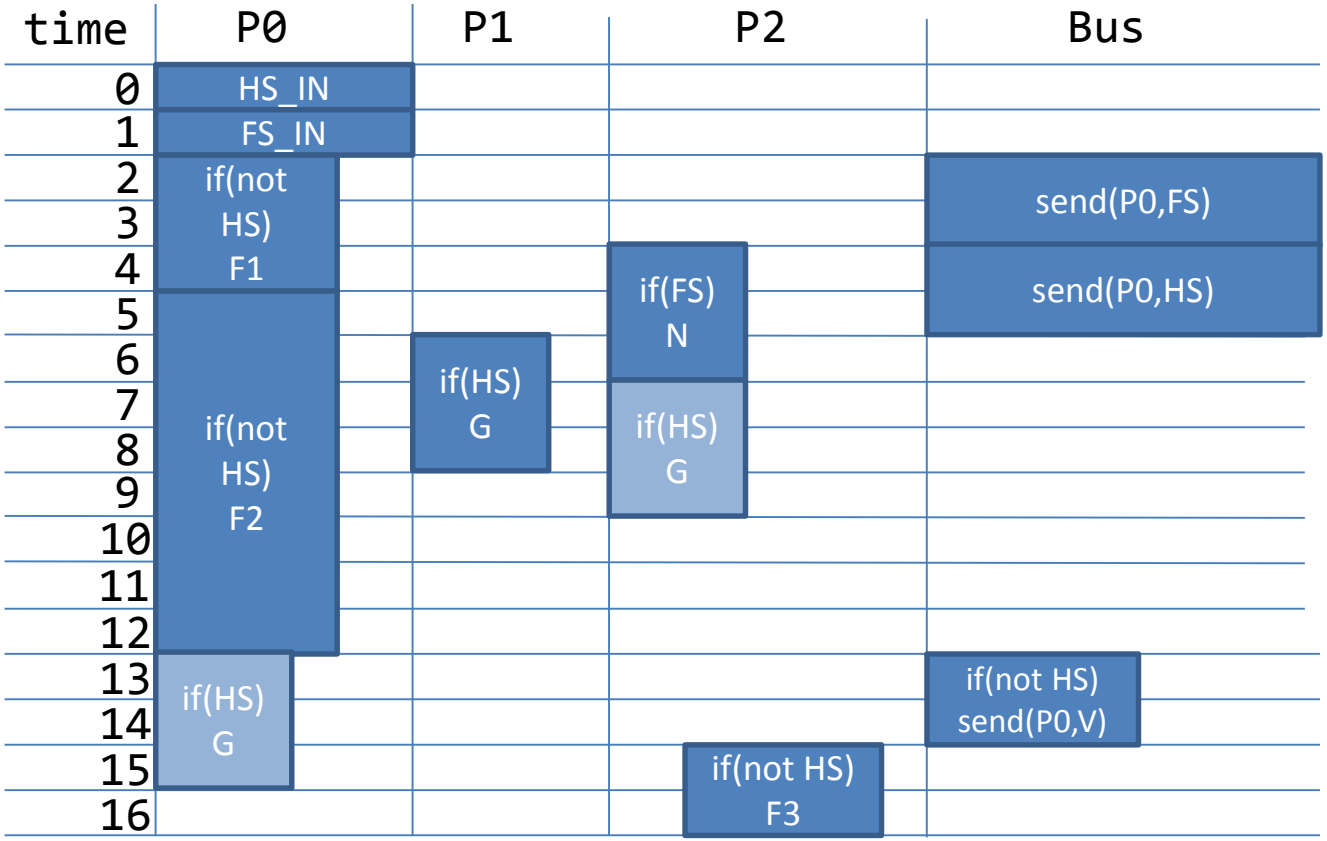
F3 can be allocated on P0, P1, or P2. It is executed much faster on P2, which compensates for the needed communication time. The allocation on P2 is therefore retained.



Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1

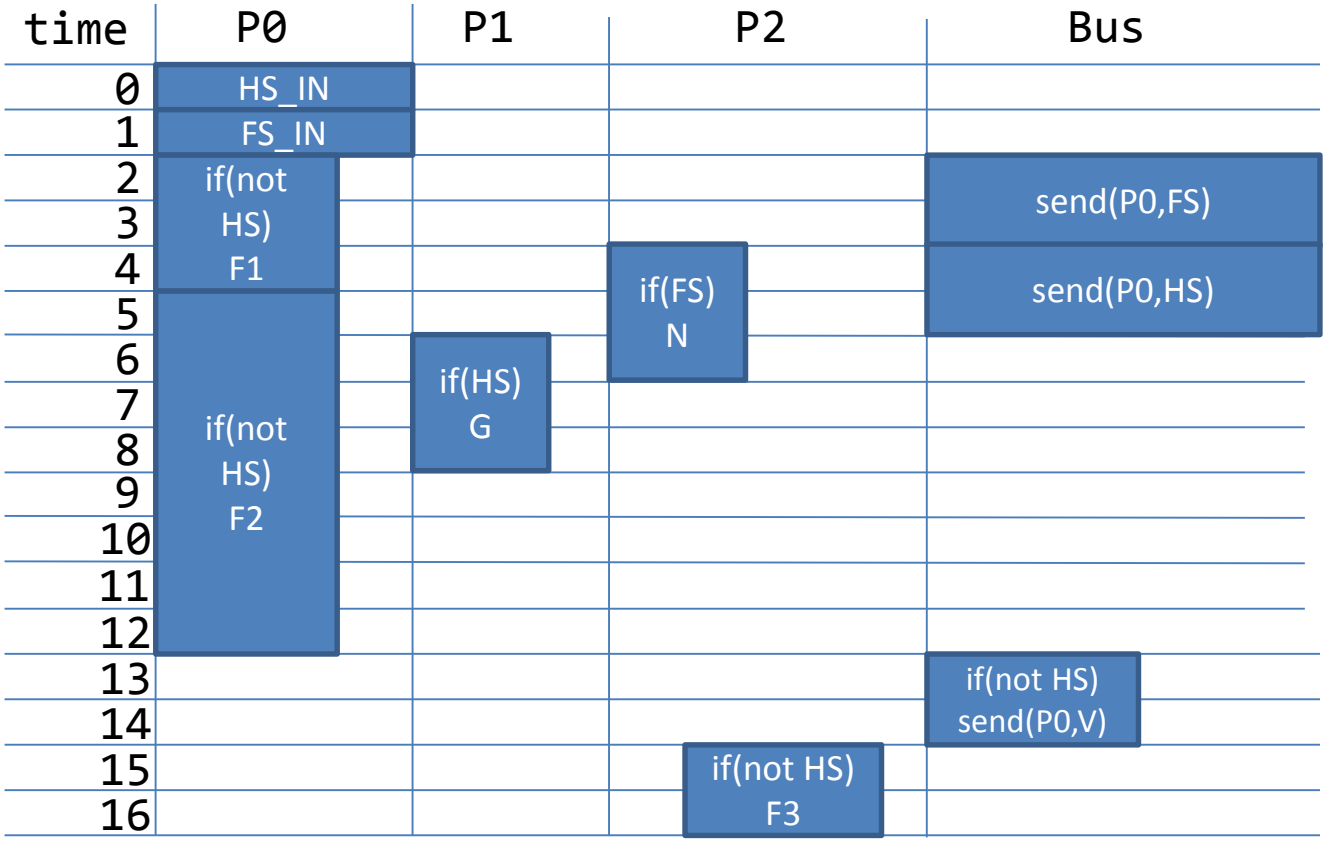
G can be allocated on P0, P1, or P2. Allocation on P1 minimizes the end date, so it is retained.



Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1

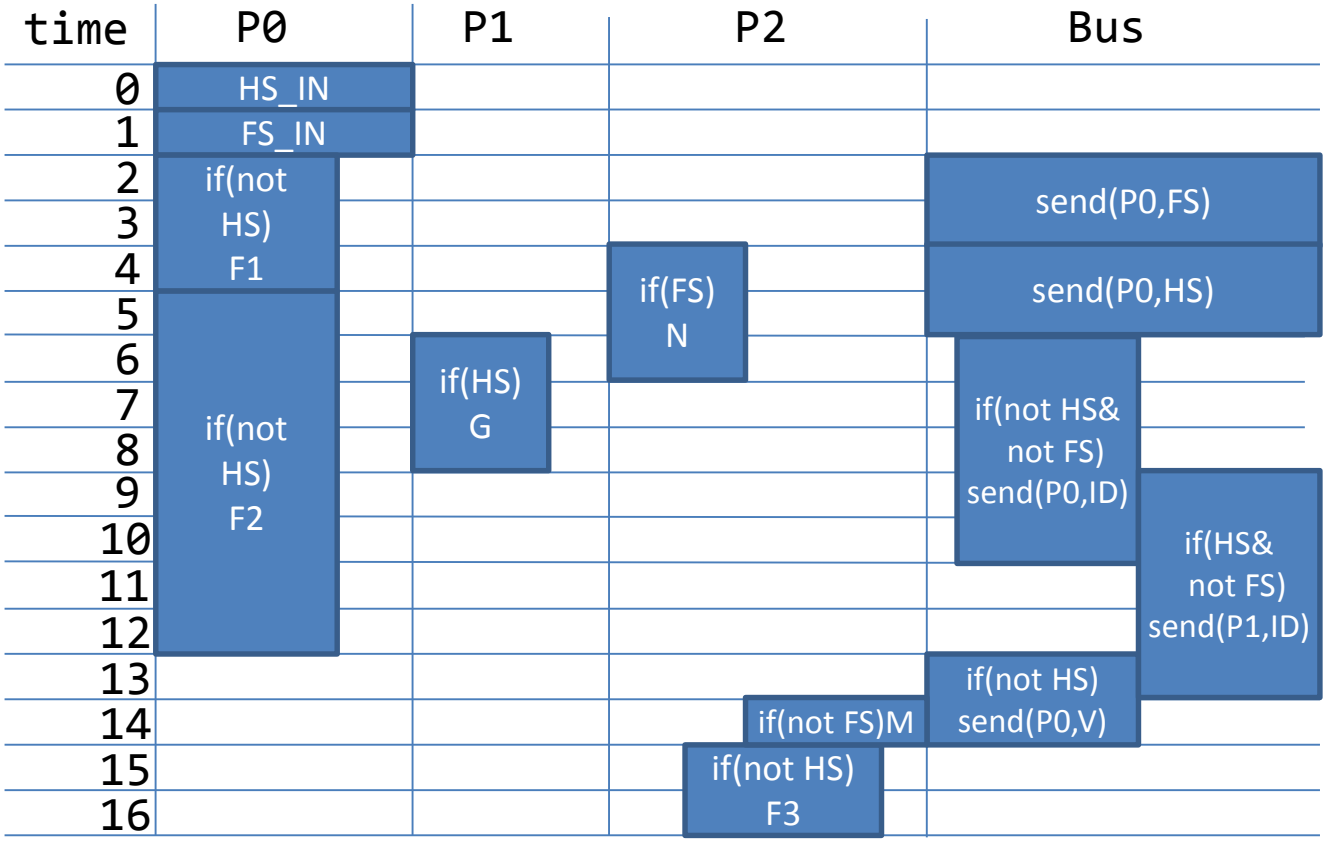
G can be allocated on P0, P1, or P2. Allocation on P1 minimizes the end date, so it is retained.



Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1

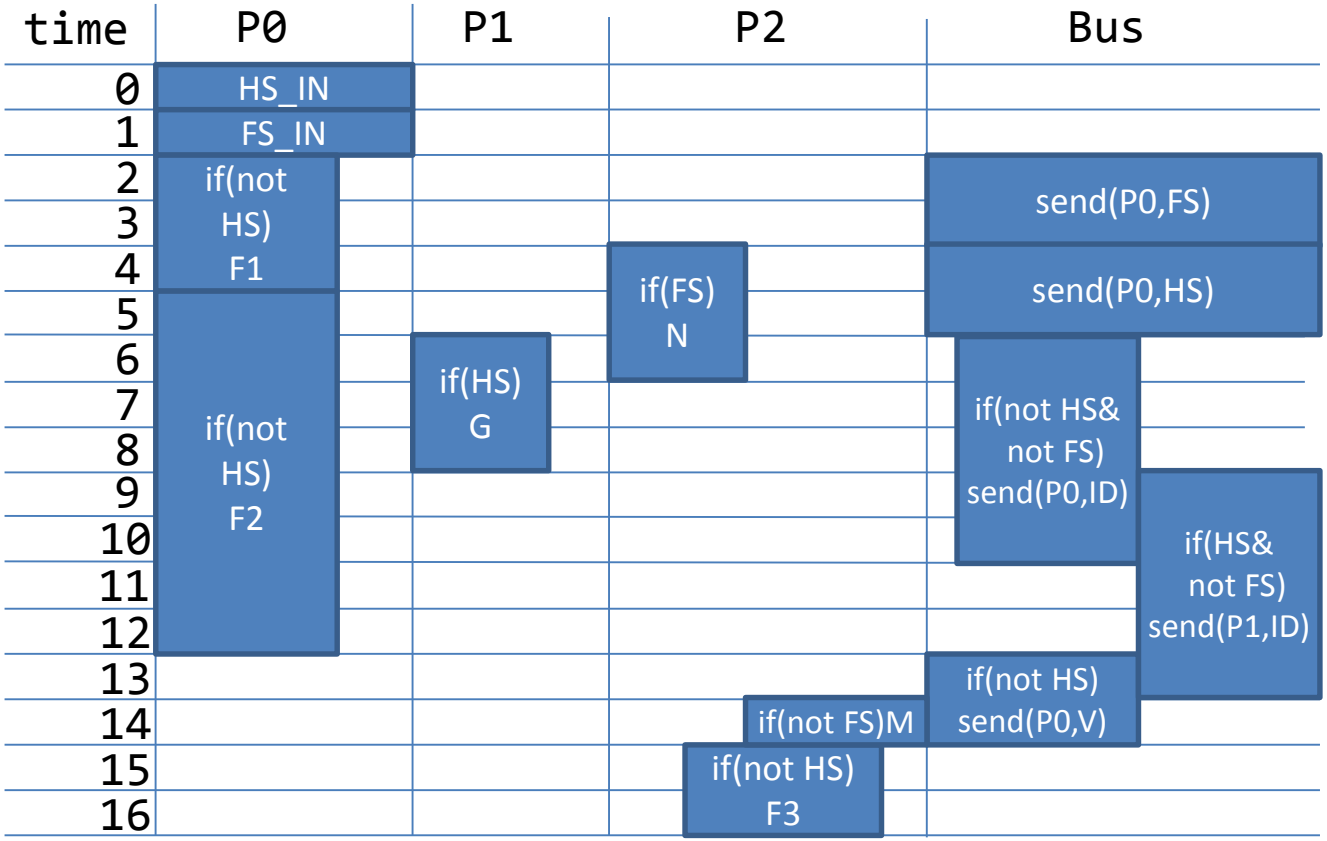
M can be allocated only on P2. Depending on the value of HS, its input ID comes from either P0, or from P1. Also this data is not needed when M is not executed, hence the execution conditions of the send operations.



Scheduling algorithm

- Step 2: allocate and schedule the blocks 1 by 1

Generated schedule:
- Latency: 17
- Throughput: 1/17



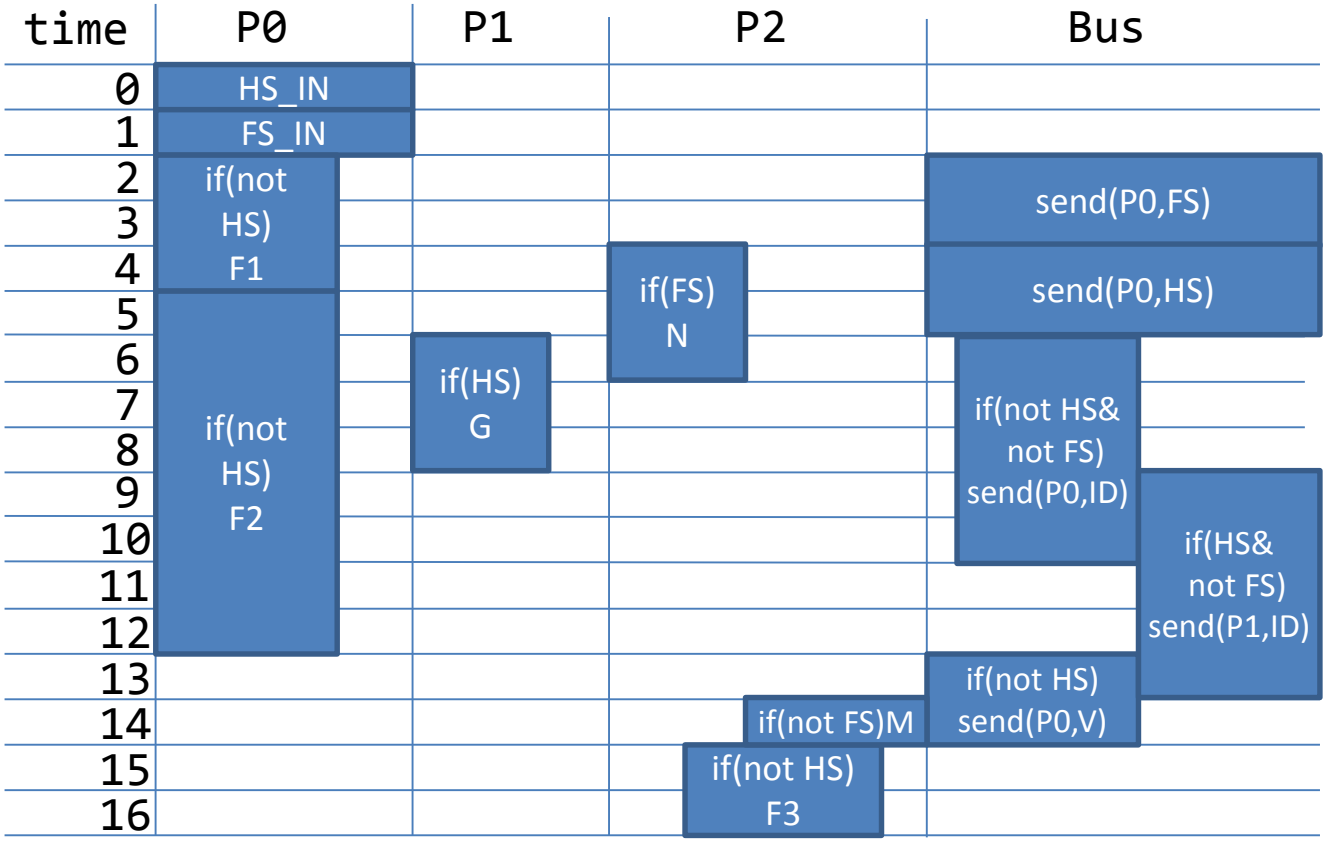
But this is not enough (1/3)

- General-purpose optimization, to reduce communications, reduce makespan, and increase throughput
  - Software pipelining of scheduling tables
    - Reduce throughput without changing makespan
    - Take into account conditional execution to allow double reservation between successive cycles
  - Safe double reservation of resources
    - Precise analysis of execution conditions

But this is not enough (1/3)

- Example: software pipelining

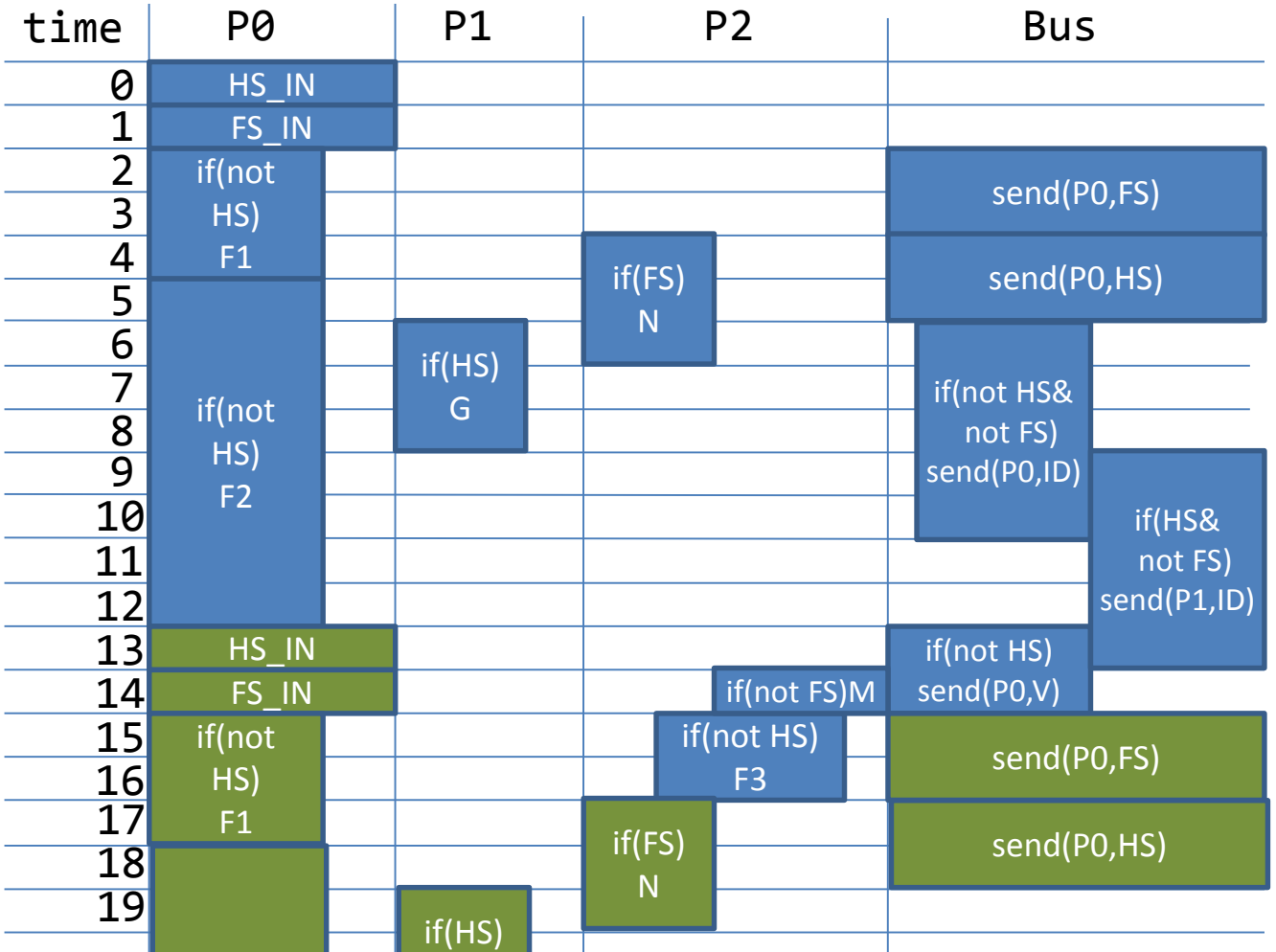
Generated schedule:
- Latency: 17
- Throughput: 1/17



But this is not enough (1/3)

- Example: software pipelining

Generated schedule:
- Latency: 17
- Throughput: 1/17

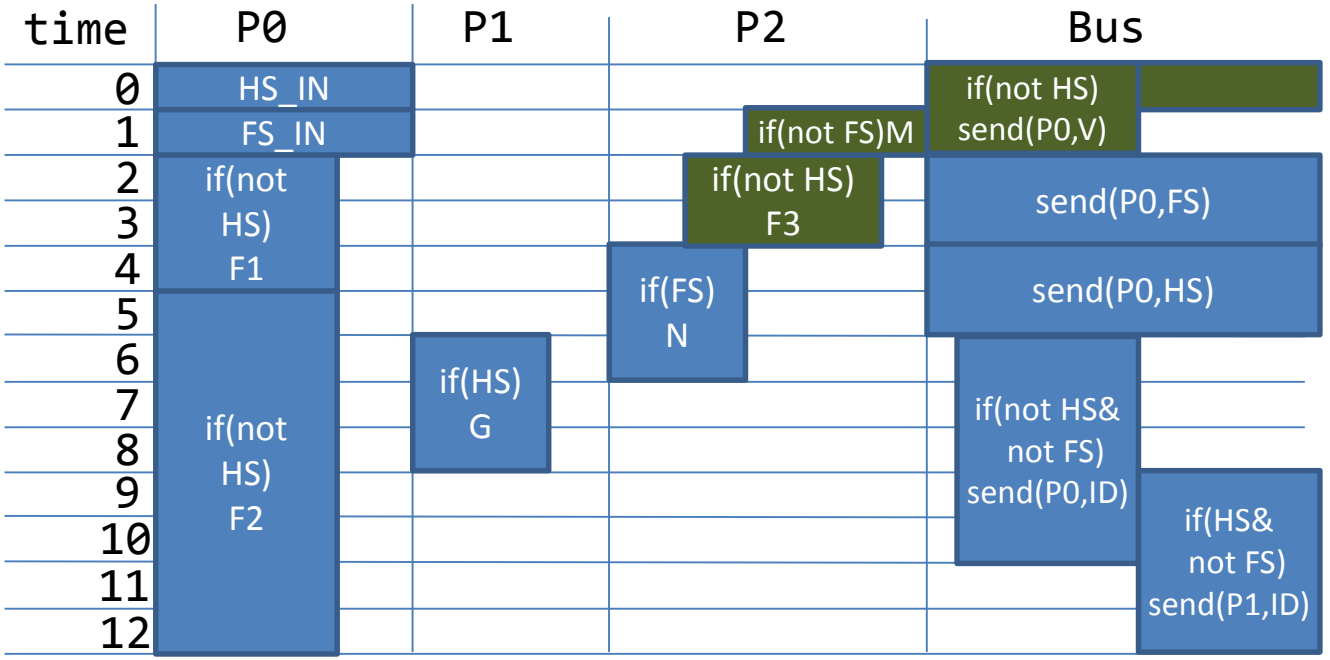


But this is not enough (1/3)

- Example: software pipelining

Generated schedule:
- Latency: 17
- Throughput: **1/13**

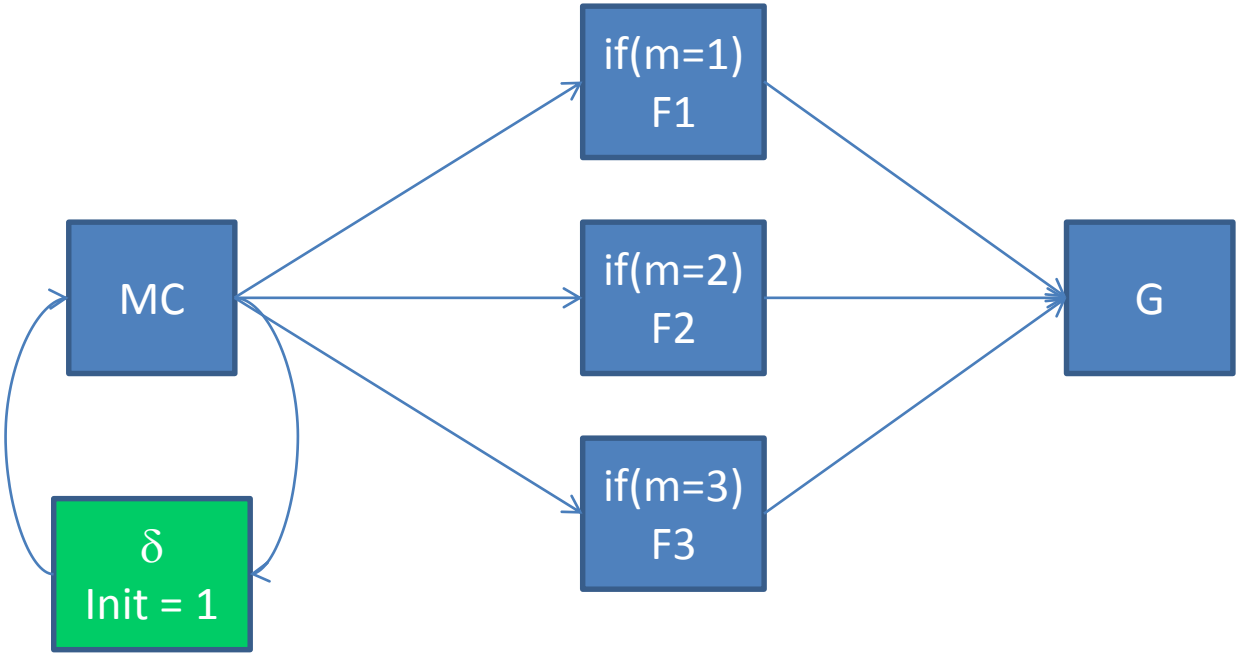
Software pipelining (classical compil. technique) can be applied.

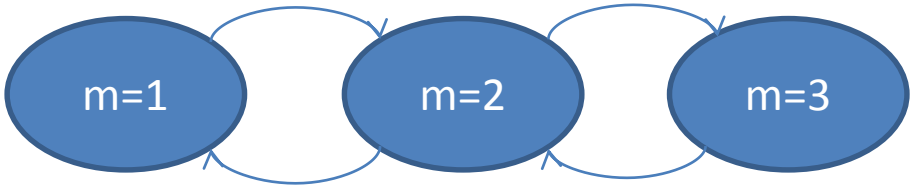


Using software pipelining required some modifications to classical algorithms:
- Bi-criteria: latency first, throughput second
- Improved predication handling (conditions between cycles, no delay slot needed)

But this is not enough (1/3)

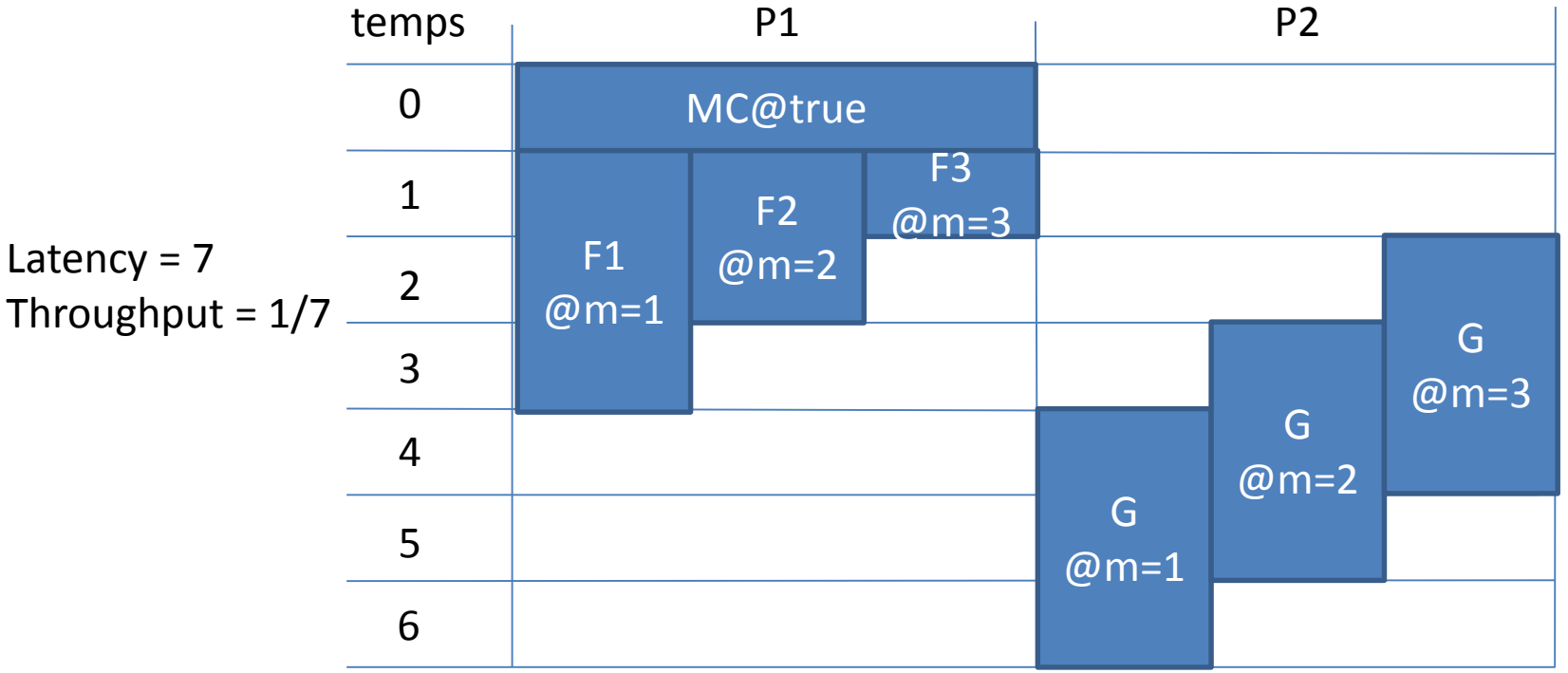
- Application





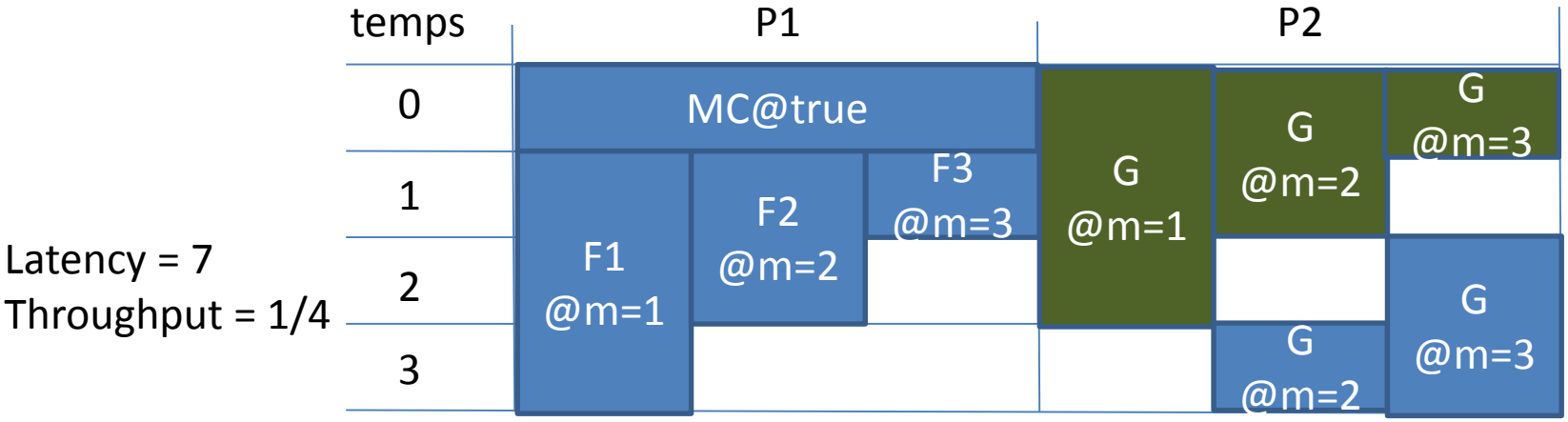
But this is not enough (1/3)

- Latency-optimizing scheduling



But this is not enough (1/3)

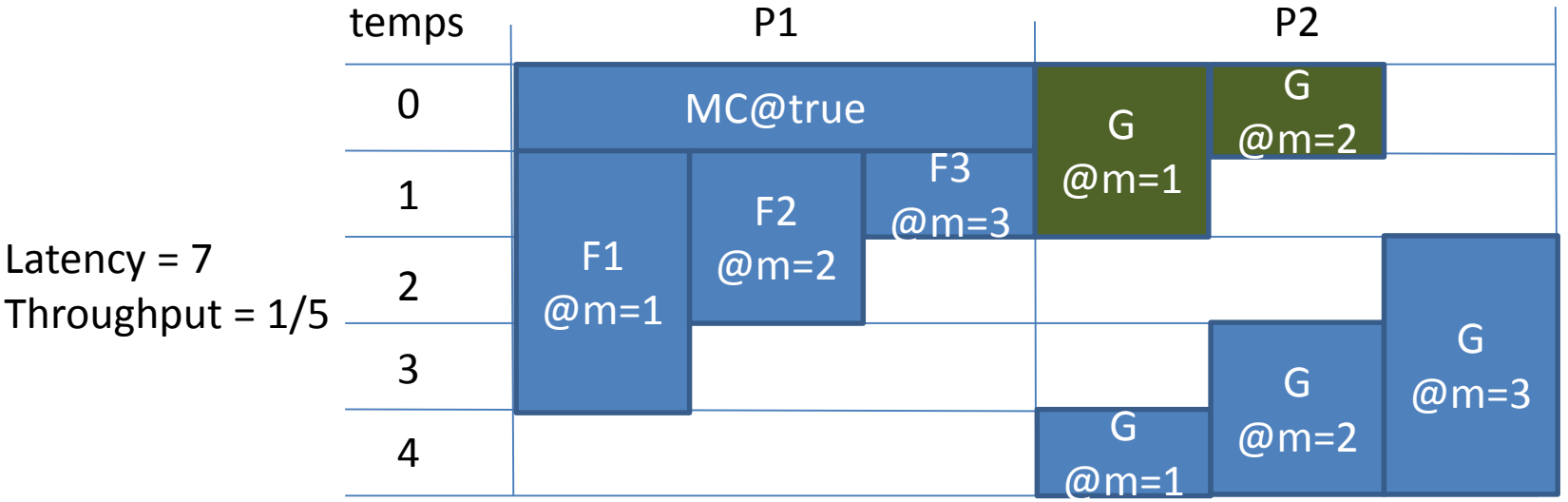
- Pipelined scheduling



  - Using mode transition information

But this is not enough (1/3)

- Pipelined scheduling



  - Without using mode transition information

But this is not enough (1/3)

- Software pipelining => memory replication
  - Rotating registers
  - Conditional execution => not clear which version of a register to access
    - Need an indirection table, dynamically updated
- Real-time guarantees
  - Worst-case analysis
  - Account for artefacts
    - E.g. rotating register operations

But this is not enough (2/3)

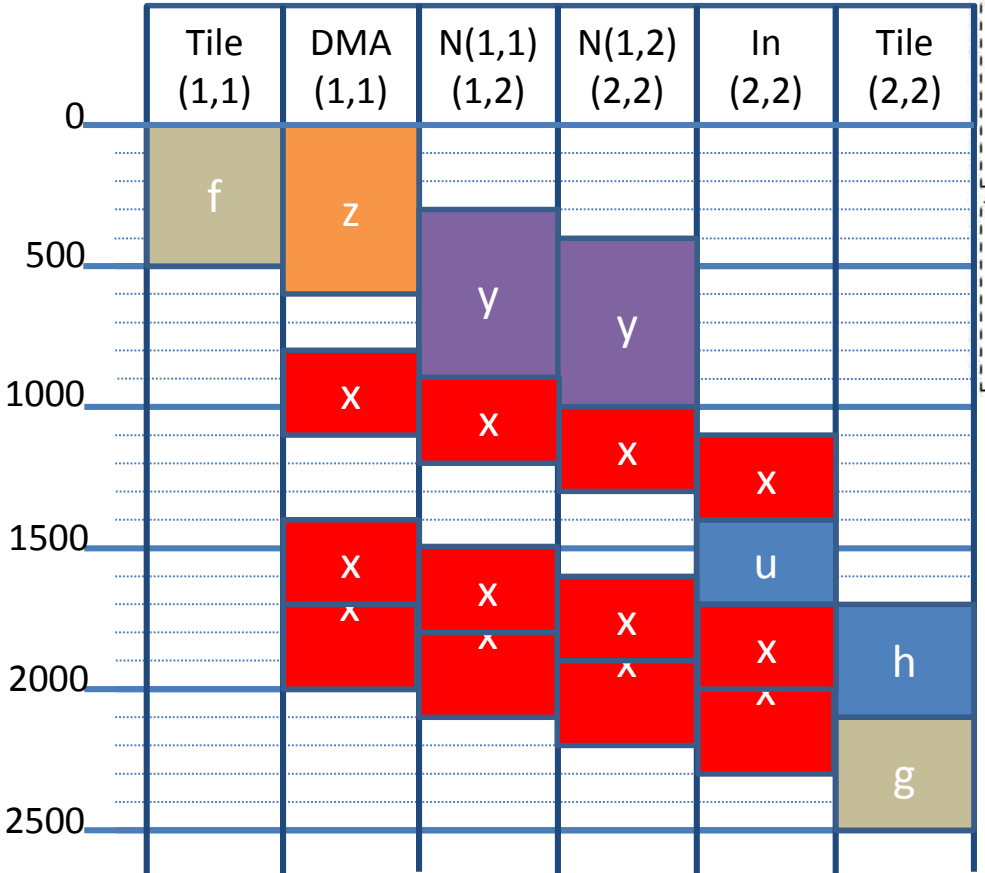
- Time triggered applications
  - Target: ARINC 653-compliant OSs, time-triggered comm.
    - Synthesis of inter-partition communication code (and considering it during scheduling)
  - Non-functional properties taken as input:
    - ARINC 653 partitioning (all or part or none, application and platform)
    - Real time (release dates, deadlines – can represent end-to-end latency)
    - Preemptability of each task
    - Allocation requirements
  - Optimizations:
    - Deadline-driven scheduling (to improve schedulability)
    - Minimize the number of tasks at code generation (at most one per input acquisition point)
    - Minimize the number of partition changes
  - Hypotheses: The scheduler and I/O impact on WCETs can be bounded.

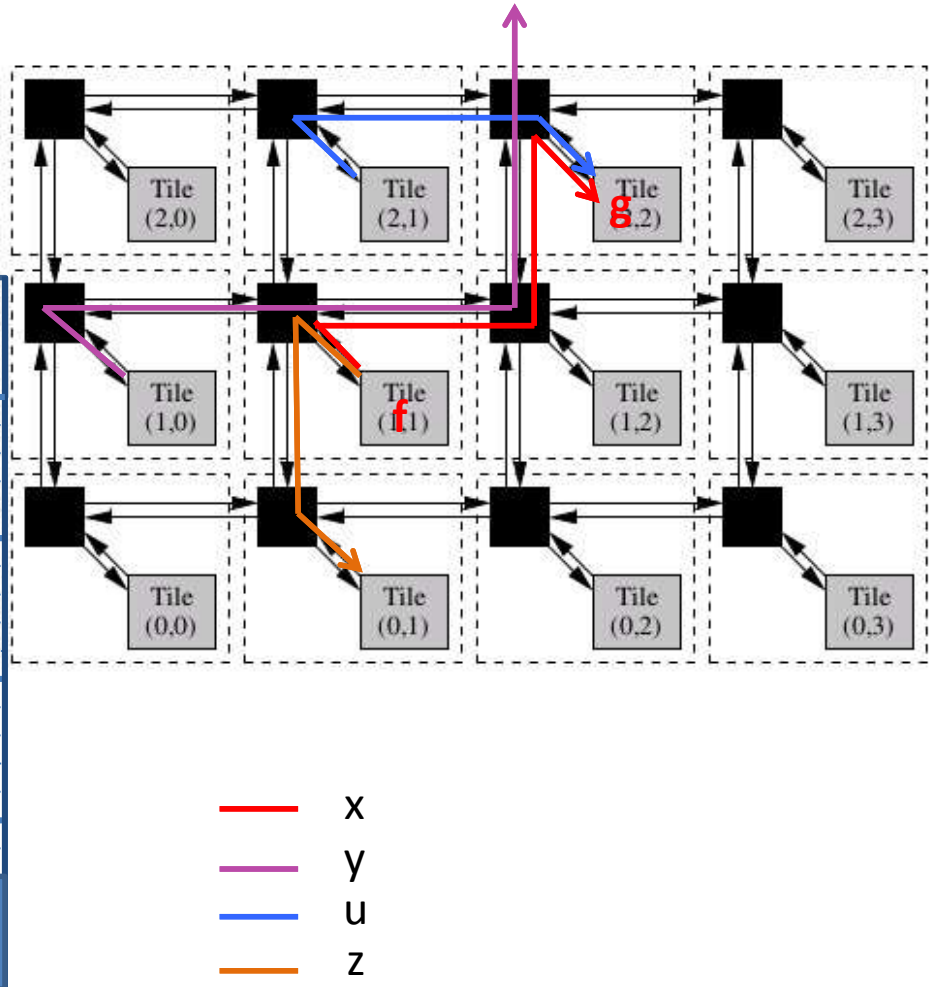
But this is not enough (3/3)

- Many-core (bare metal)
  - Sound architecture abstraction layer with support for efficiency
    - Precise platform modeling:
      - NoC: one resource per DMA and NoC arbiter
        - » Wormhole scheduling synchronizes schedules along the path of each packet
      - Memory banks are resources
        - » Reserve them for data, code, and stack **to avoid contentions**
    - **WCET analysis for parallel code**
      - Considers synchronization overheads
  - Synthesis of communication and synchronization code (lock-based) on the processors
    - Account for communication/synchronization initiation costs
  - Precomputed preemptions for communications
    - Account for preemption costs

But this is not enough (3/3)

- Synchronized & preemptive comm.





Other related results (1/3)

- Many-core platform with support for efficient predictability
  - Principle: avoiding contentions through mapping
    - No shared caches
    - Hardware locks for mutual exclusion during RAM access
  - Expose NoC arbitration to programming
    - **Efficiency bottleneck is in both CPUs and interconnect, and similar arbitration is required of both (in various contexts), so they should provide the same level of control.**
    - General-purpose:
      - No change in processor programming (gcc)
      - Not programming the NoC arbiters maintains functional correctness
    - Small hardware overhead
    - Automatic mapping (global optimization) -> good results !

Other related results (2/3)

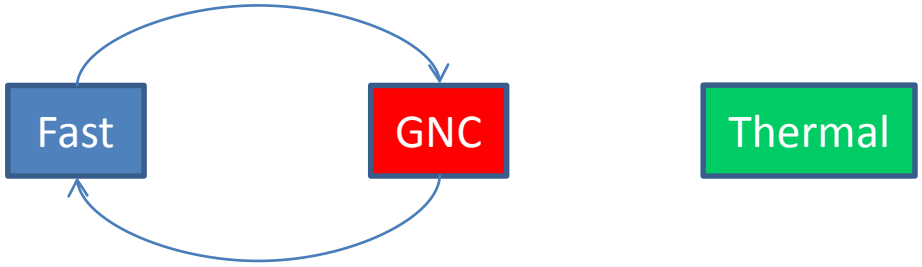
- Heuristics vs. exact methods (SAT/SMT/etc.)
  - Recent constraint solvers (Cplex, Gurobi, Yices2, etc.) have largely improved performance
  - Question: Can they solve scheduling problems ?
  - Answer: Yes, up to a certain size, which largely depends on problem complexity, problem type (schedulability vs. optimization), system load.
    - From a few tasks (optimization, preemptive, multi-periodic) to more than 100 tasks.
  - Conclusion: **Developing heuristic methods still has its use, especially for complex problems like ours**

Other related results (3/3)

- Automatic delay-insensitive implementation of synchronous programs
  - Objective: separation of concerns between functional and non-functional aspects
    - Optimal synchronization protocols ensuring functional determinism
  - Characterization of delay-insensitive synchronous components (weak endochrony)
    - Set theoretical limits
  - Weak endochrony checking
    - New representation of the behavior of concurrent systems by means of sets of atomic behaviors
  - Synthesis of delay-insensitive concurrent implementations

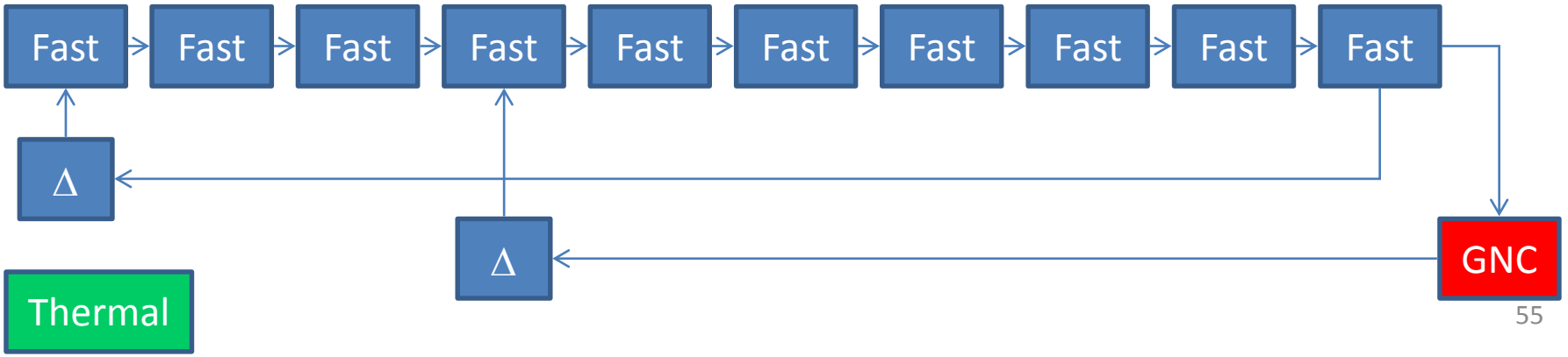
Avionics GNC application

- Simplified version:
  - 3 tasks T_{Fast} ,T_{GNC} et $T_{thermal}$
  - 3 partitions P_{Fast}, P_{GNC} et $P_{thermal}$
  - WCETs: 40ms, 200ms et 100ms
  - Periods: 100ms, 1s, 1s



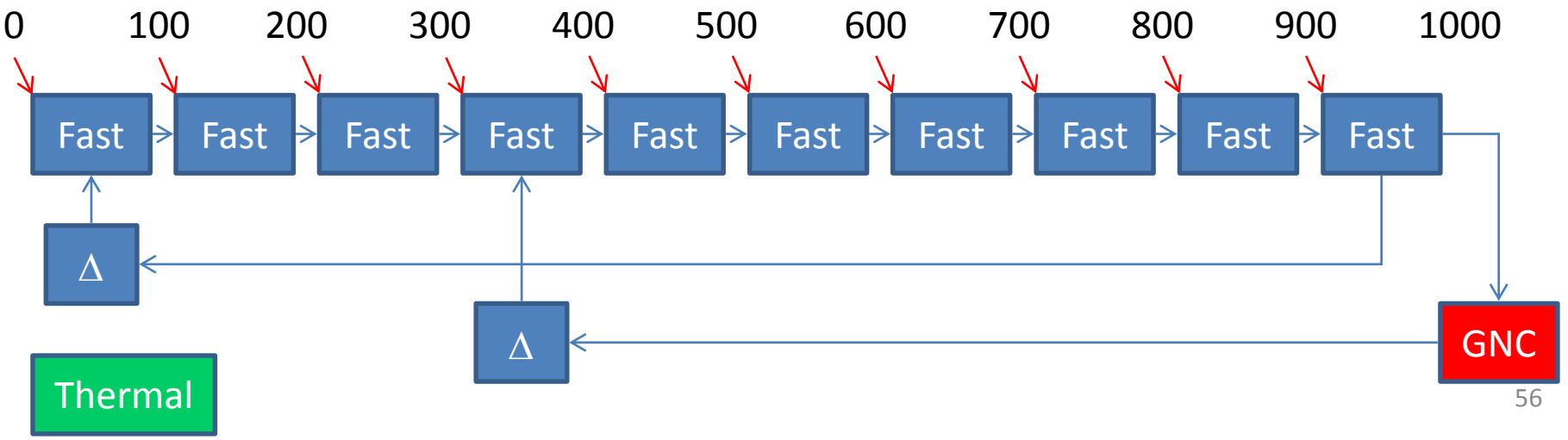
Avionics GNC application

- Simplified version:
  - 3 tasks T_{Fast} ,T_{GNC} et $T_{thermal}$
  - 3 partitions P_{Fast}, P_{GNC} et $P_{thermal}$
  - WCETs: 40ms, 200ms et 100ms
  - Periods: 100ms, 1s, 1s
  - Hyperperiod expansion : MTF = 1s



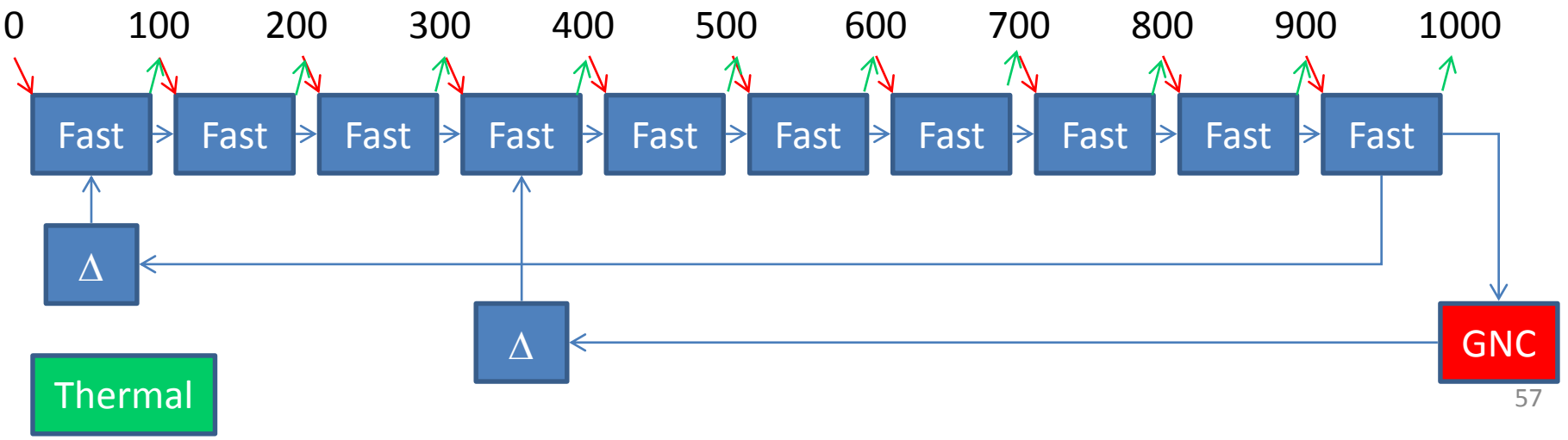
Avionics GNC application

- Real time requirements
  - T_{fast} reads inputs arriving periodically in an infinite buffer



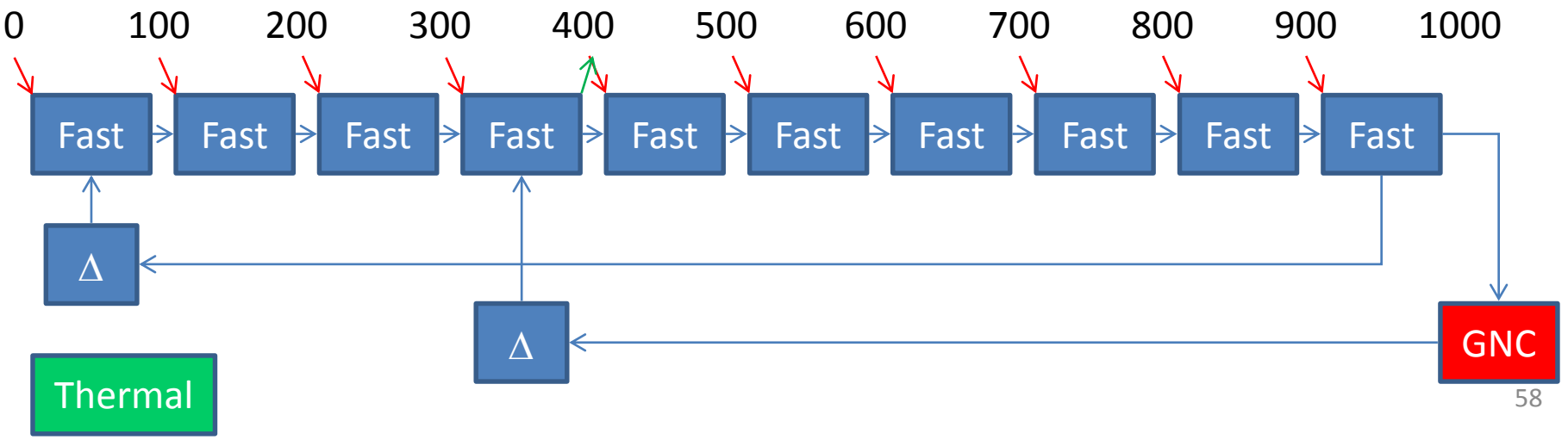
Avionics GNC application

- Real time requirements
  - T_{fast} reads inputs arriving periodically in a buffer of size 2



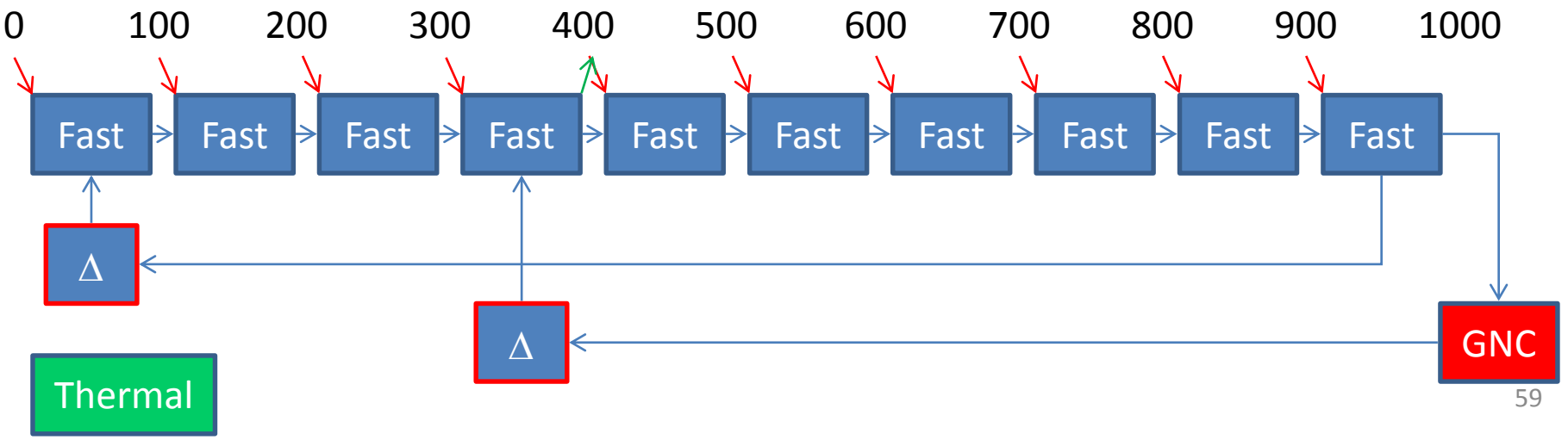
Avionics GNC application

- Real time requirements
  - Infinite buffers
  - « A full flow formed of 10 instances of T_{fast} followed by an instance of T_{GNC} and then by an instance of T_{fast} should take less than 1400 ms »



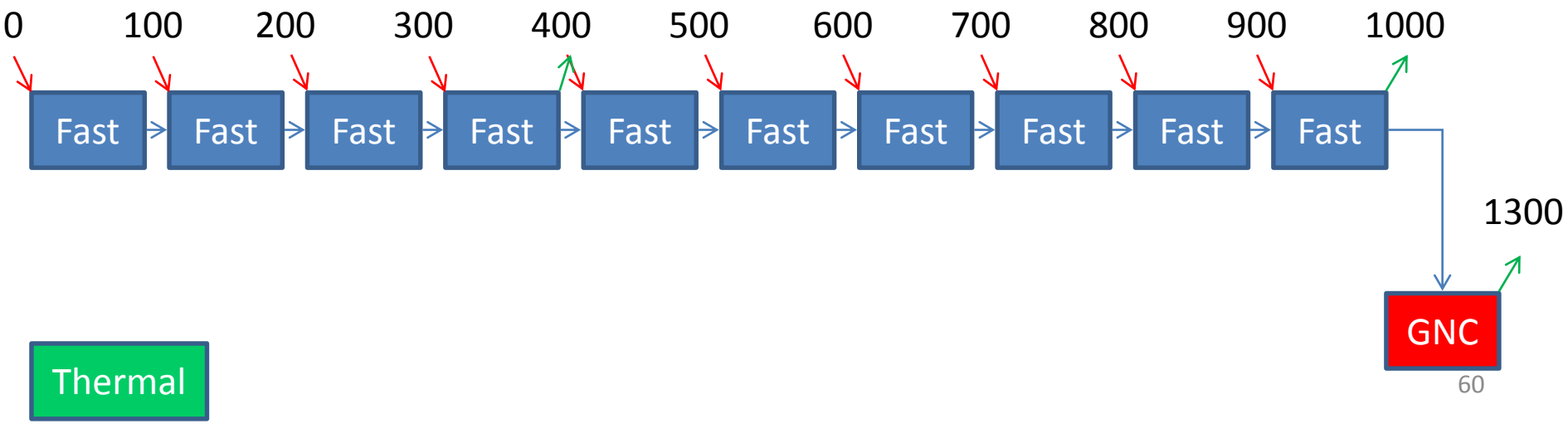
Avionics GNC application

- Step 1 : remove delayed dependencies
  - Replace them by timing barriers
  - Heuristic approach (sub-optimal)



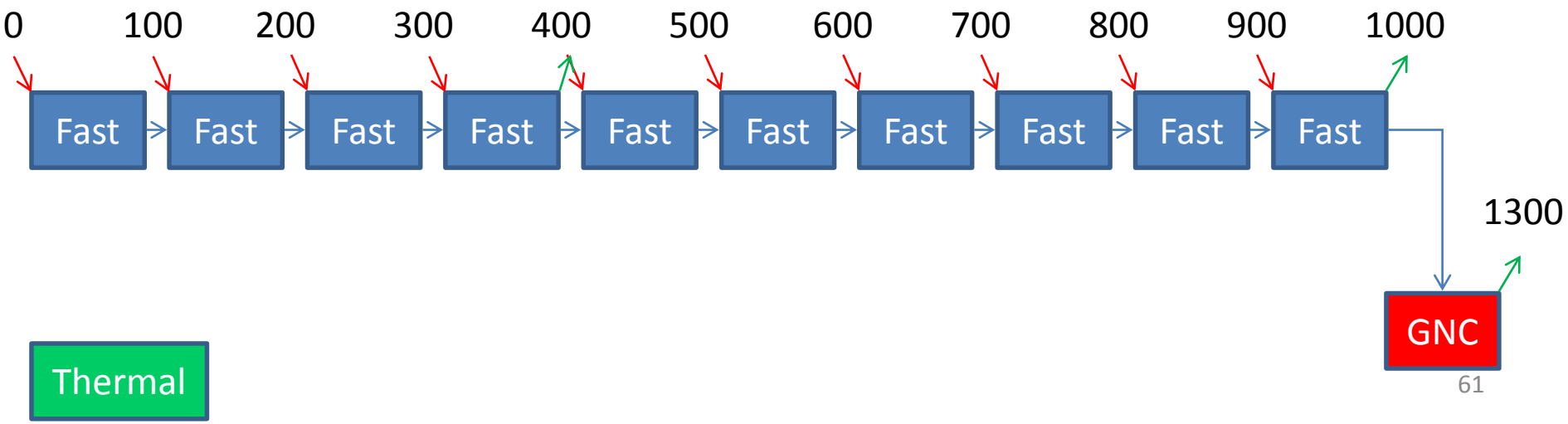
Avionics GNC application

- Step 1 : remove delayed dependencies
  - Replace them by timing barriers
  - Heuristic approach (sub-optimal)



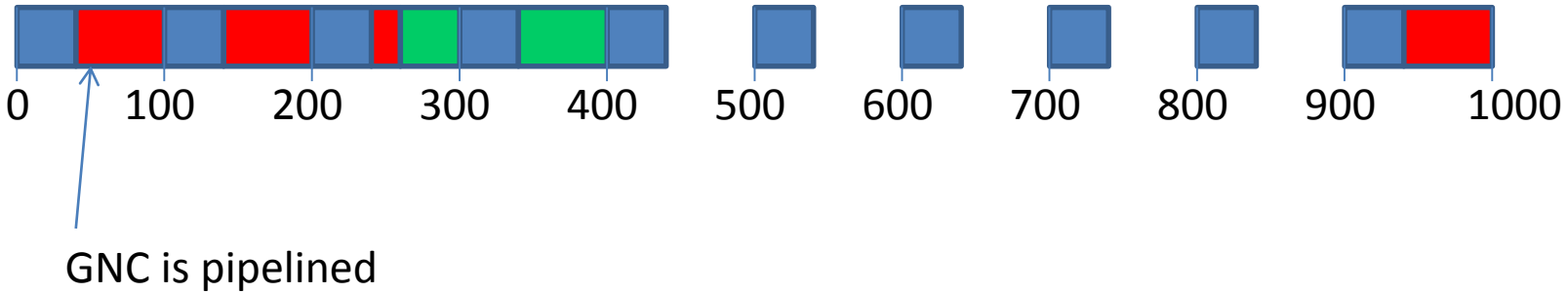
Avionics GNC application

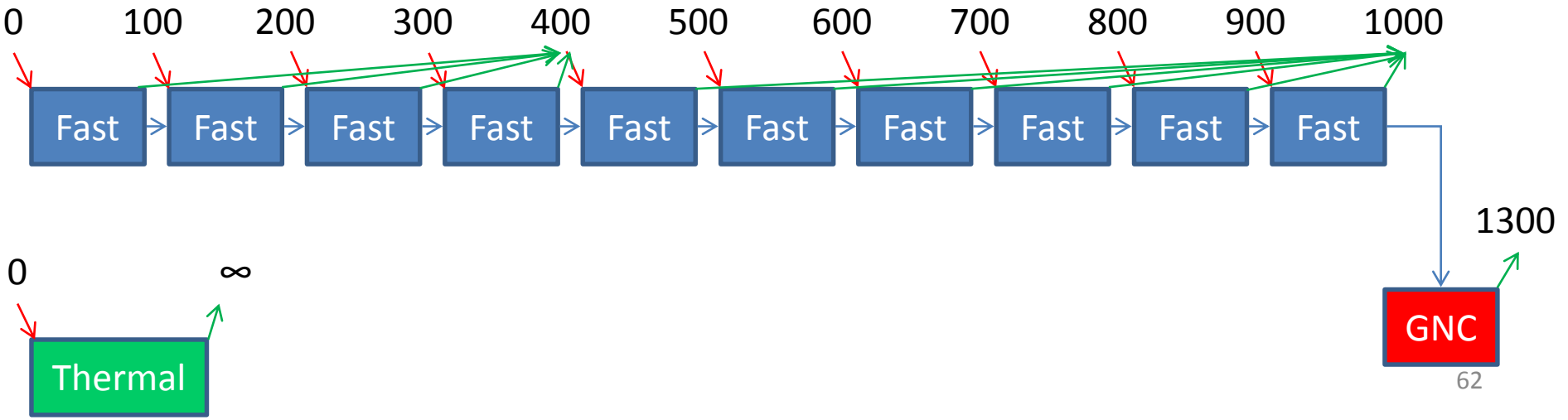
- Step 2 : off-line scheduling
  - Allocate and schedule blocks one by one (no backtracking)
  - Deadline-driven
    - Of all tasks that can be executed, consider the one with earliest deadline
    - Deadline = minimum of all successor deadlines



Avionics GNC application

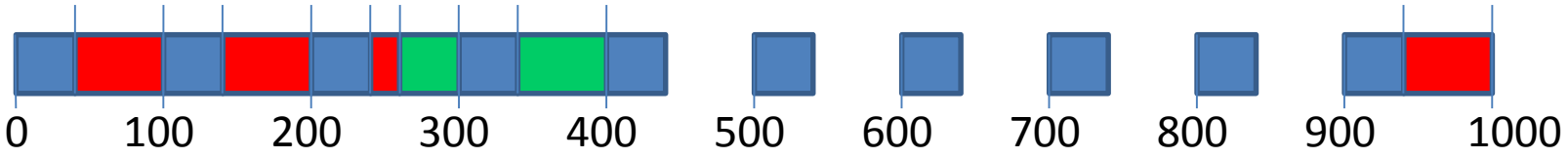
- Step 2: off-line scheduling



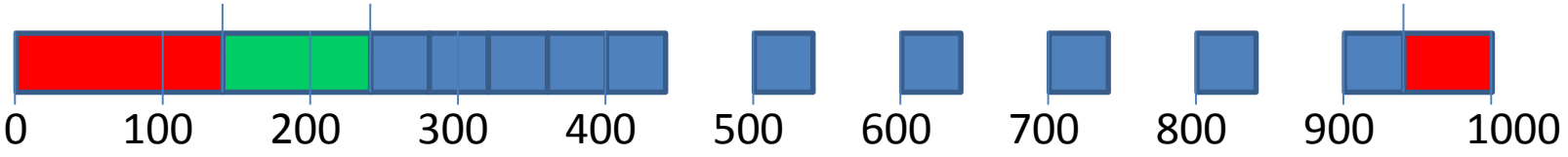


Avionics GNC application

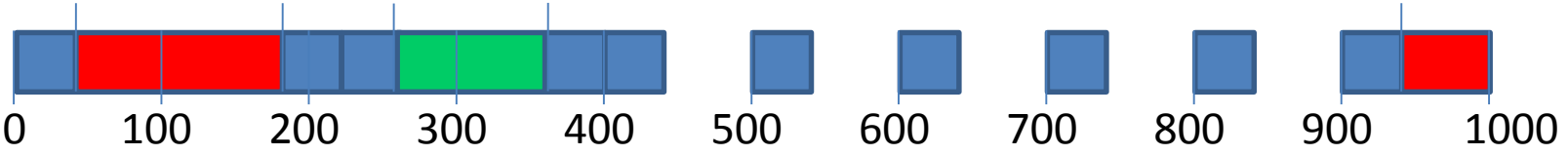
- Step 3: post-scheduling optimization



  - Deadline-driven scheduling routine
    - Many context/partition changes
    - Reduce their number by moving reservations subject to timing and data dependency requirements
      - Low complexity
  - Possible results



Infinite buffer



Buffer of size 3

Avionics GNC application

- Full application
  - 4 processors, 1 broadcast bus
  - 13 tasks before hyper-period expansion
  - 7 partitions
  - 10 end-to-end latency constraints
  - Result:
    - Processor loads: 82%, 72%, 72%, 10% (last one for telemetry)
    - Bus: 81%

Conclusion (1/3)

- **Full-fledged real-time systems compilation is feasible**
  - At least for certain application classes
  - Ensures correctness and efficiency
  - Allows a trial-and-error design approach for complex embedded systems
- Simple theoretical and algorithmic principles
  - Independence, isolation, list scheduling, timetables, pipelining…

Conclusion (2/3)

- But no not mistake a list of simple principles for a compiler
  - Integration requires careful adaptation of the concepts and techniques
  - Match the needs of realistic applications to give **good results**
    - Good resource use
    - Good timing precision
      - Few percent difference between guarantees and actual execution for certain many-core applications.
  - Lopht: not a scheduling toolbox, nor a classical compiler
    - Hence the title « **real time systems compiler** »

Conclusion (3/3)

- Major difficulties
  - Few benchmarks/case studies
    - Sharing is difficult (technical, legal, will issues)
  - Publication
    - Cross-domain, everybody sees it as interesting « for the other domain » (sort of NIMBY syndrome)
  - Sharing between various targets
    - No general platform definition language

Future work (1/2)

- Short/medium term (ongoing):
  - Execution platform modeling
  - Provide compiler correctness guarantees/proofs
  - Improve treatment of multi-periodic applications
    - Trade-off between scheduling freedom (and thus efficient resource use) and compactness of representation (architectural limits, legibility)
  - New platforms
    - TTEthernet – taking into account platform-related tools
    - Kalray MPPA – Generate code over the Kalray hypervisor, NoC handling
  - New case studies for/from the industry

Future work (2/2)

- Longer term:
  - Improve embedded systems mapping by adapting/combining models and techniques from compilation, real-time scheduling, and synchronous languages
    - Compilation: Code replication, Loop unrolling…
    - Real-time scheduling: CRPD computation…
    - Synchronous languages: n-synchronous clock calculus…