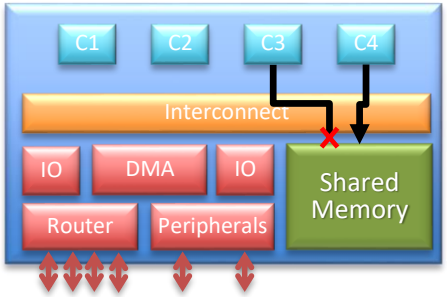
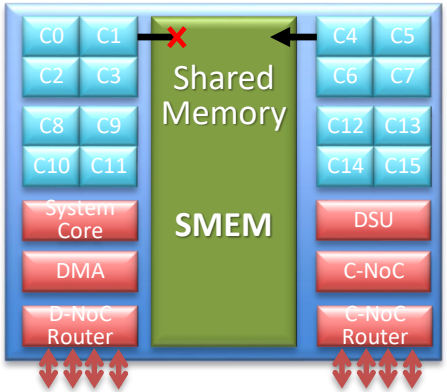
Efficient fine-grain parallelism in shared memory for real-time avionics

P. Baufreton – Safran
V. Bregeon, J. Souyris – Airbus
K. Didier, **D. Potop-Butucaru**, G. Iooss – Inria

Critical real-time on multi-/many-cores

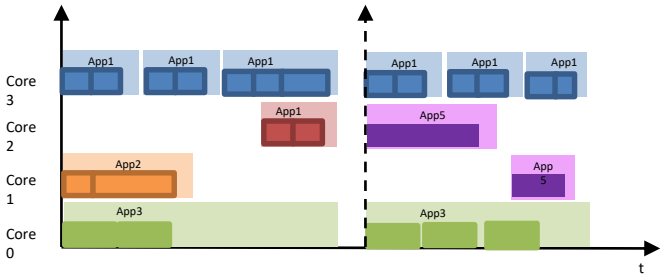
- All the single-core problems plus:
  - Significantly more concurrency
    - More sources of interferences
  - Making the parallelization decisions
    - And more complicated memory allocation, etc.
- Ensuring safety is paramount
  - Time/space isolation facilitates the demonstration of certain properties
- Ensuring efficiency
  - Bad implementation decisions -> poor performance
    - If you get 1.2x acceleration on two cores, then maybe it's not worth it…
    - **Too much isolation -> poor performance!**

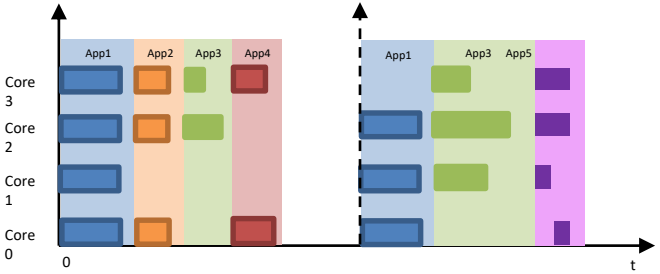




Critical real-time on multi-/many-cores

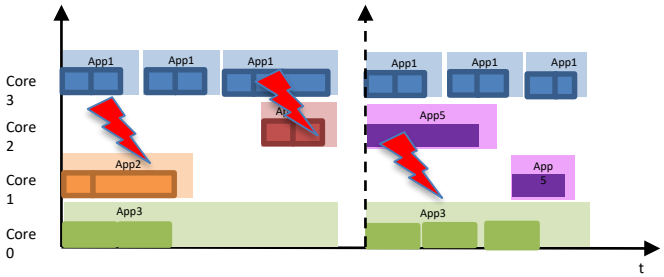
- IMA = Integrated Modular Avionics
  - Partition = dual concept
    - Piece of (multi-task) software
    - Resources statically allocated to this software
  - Time-Space Partitioning (TSP)
    - A partition must never over-step its resource allocation
- CAST-32A – Avionics recommendations for multi-core implementation
  - Maintains strict TSP requirement between partitions: « Robust Resource and Time Partitioning » is difficult





Critical real-time on multi-/many-cores

- Current approach – natural extension of single-core practice
  - One partition executes on only one core
    - Often corresponding to re-usable modules
  - Advantage: modularity in development
  - Disadvantages:
    - Performance – due to lack of parallelization inside partitions and due to TSP between partitions
    - Difficult to demonstrate Robust Resource and Time Partitioning on common multi-core platforms
      - Interferences between partitions running in parallel
      - Requires HW resource partitioning (e.g. caches, RAM, I/O, etc.)

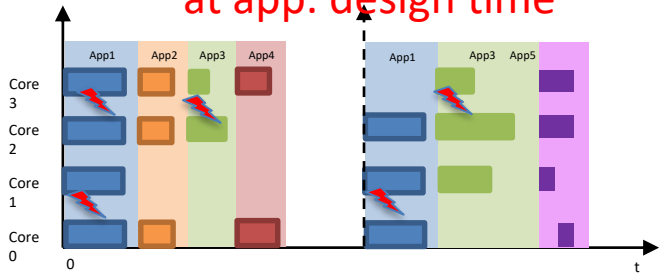


Interferences known only at integration time

Critical real-time on multi-/many-cores

- Possible solution: Parallelize partitions
  - Fixed resource envelopes (Cores, memory banks)
  - Advantage:
    - If all partitions are parallelized on all cores, **classical IMA TSP between partitions**
      - Empty caches, reset shared devices
    - No time or space isolation required inside the partition
  - Difficulty: efficient parallelization is not easy
    - Concurrent resource allocation = NP-complete
      - But efficient heuristics exist
    - Timing analysis of parallel code is difficult
      - Interferences due to the access to shared resources
      - **Time/space isolation properties are often used to facilitate timing analysis**, reducing efficiency

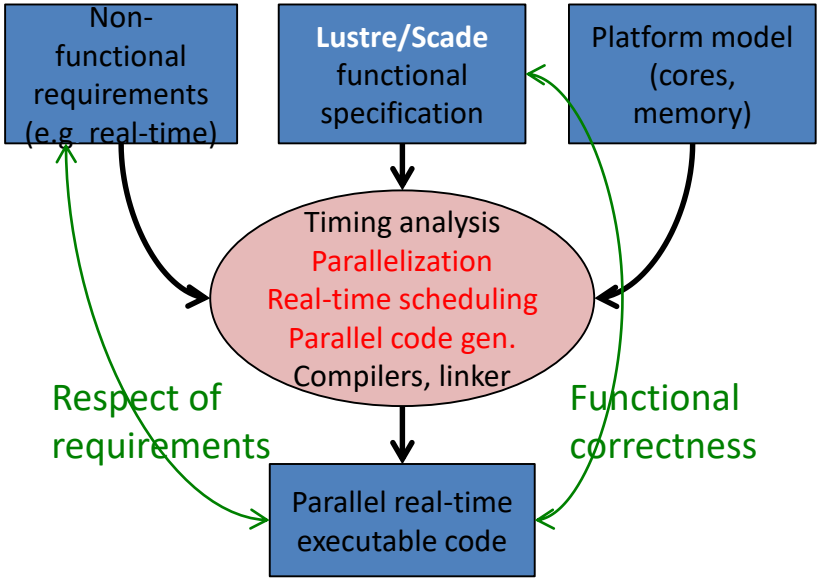
Interferences known at app. design time



Our previous work: LoPhT

[ACM TACO'19]

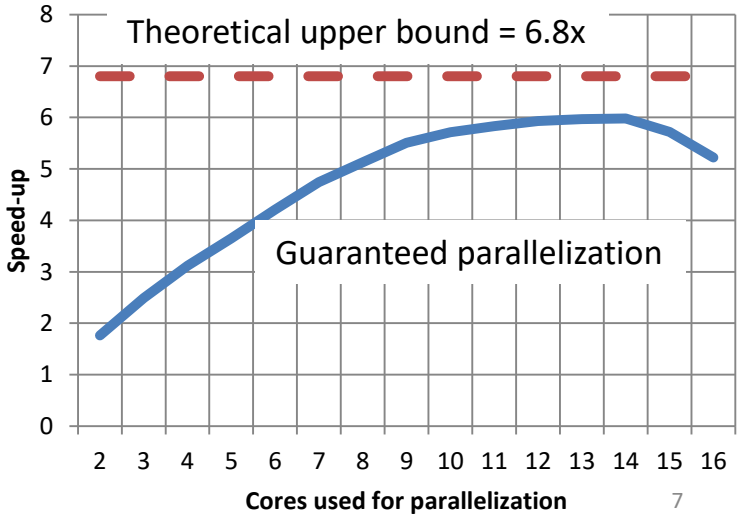
- Efficient parallelization of one partition
  - Allow interferences and control them -> better resource sharing/usage
  - Guarantee respect of real-time requirements
  - Scalable
  - Efficient:
    - Low memory footprint
    - Low synchronization overhead
    - Efficient scheduling
    - Memory allocation to minimize cache misses and interferences



Our previous work: LoPhT

[ACM TACO'19]

- Two large use cases:
  - Flight controller (>5k nodes, >36k variables)
    - 5.17x speed-up on 8 cores for the flight controller (upper bound: 6.8x)
  - Aircraft engine control
    - 2.66x on 4 cores (upper bound: 2.69x)
  - Target platforms:
    - Kalray MPPA 256 Bostan compute cluster (16 cores)
    - T1042 (4 cores) ongoing work
      - Also improve sequential code generation



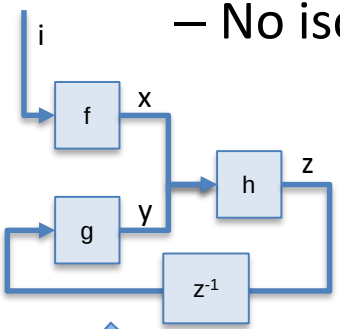
This work

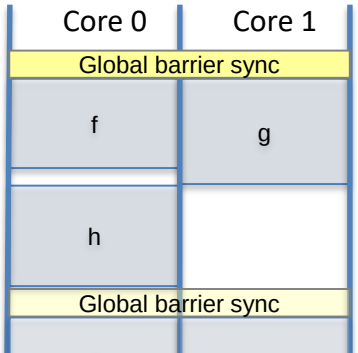
- Evaluate the efficiency cost of isolation properties
  - Use Lopht and the use cases
  - Enforce isolation properties through mapping and code generation
  - Determine the costs

  - Do not focus on very costly isolation mechanisms that are obviously not needed when parallelizing (e.g. full-fledged ARINC653-like TSP), but on those proposed in the literature/industry for the same type of application

Space isolation

- Optimized Lopht code generation
  - No isolation, one C variable per dataflow variable, all users access it






```
void* thread_cpu0(void* unused){
   lock_init_pe(0); init();
   for(;;){
      global_barrier_reinit(2);
      time_wait(3000);
      global_barrier_sync(0);

      dcache_inval();
      f(i,&x);
      dcache_flush();
      unlock(1);

      lock(0,0);
      dcache_inval();
      h(x,y,&z);
      dcache_flush();
   }
}
```

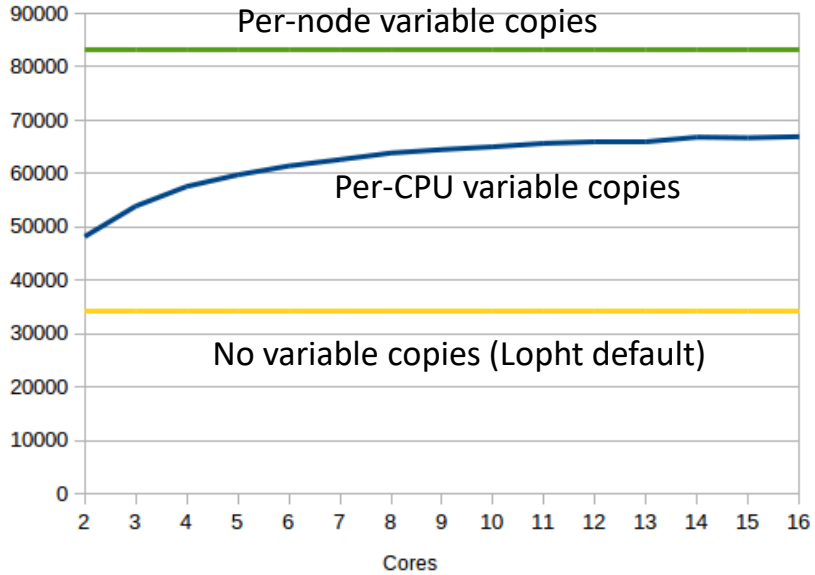
```
1  void* thread_cpu1(void* unused){
2     lock_init_pe(1);
3     for(;;){
4
5
6        global_barrier_sync(1);
7
8        dcache_inval();
9        g(z,&y);
10       dcache_flush();
11       lock(1,1);
12       unlock(0);
13
14
15
16
17    }
18 }
```

Space isolation

- Space isolation
  - Between threads/cores
    - Each one has a separate copy of the variables it uses
    - Explicit copy operations to transfer values from one core to another
  - Between tasks/nodes
  - Advantage:
    - In conjunction with memory allocation policies it facilitates timing analysis, error isolation
      - e.g. One memory bank per core, computations only access local bank
  - Disadvantages:
    - Memory footprint
    - Copy operations overhead
    - Error isolation is not required inside a partition! (over-engineering)

Space isolation

- Space isolation – memory footprint
  - Flight controller application – communication vars.



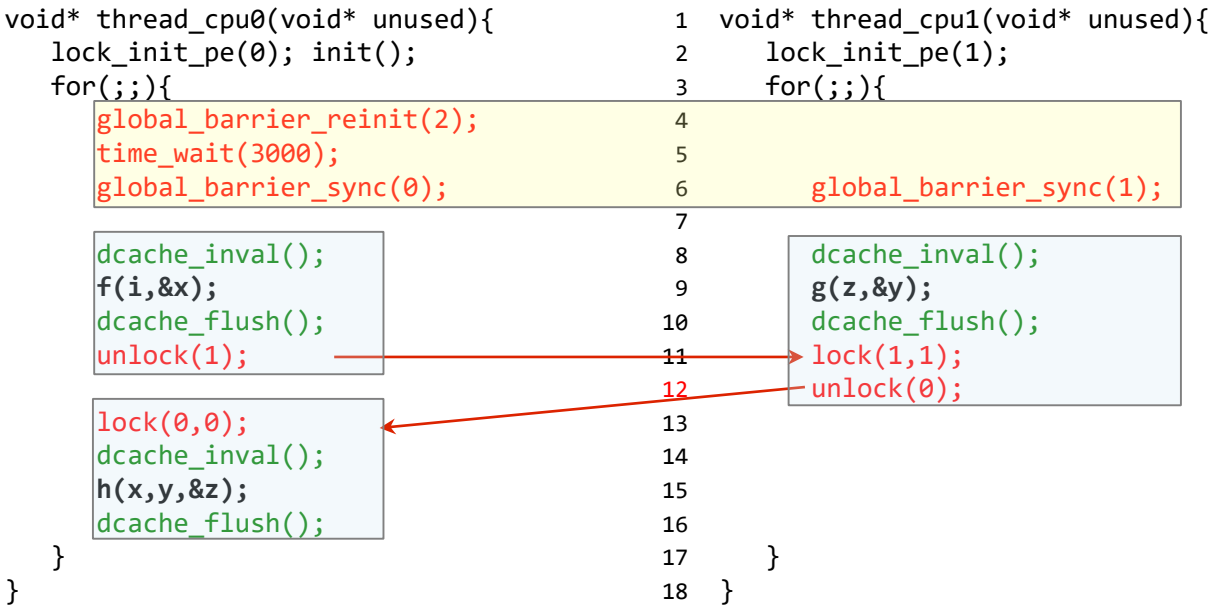
  - Copy operations (one per variable copy)

Time isolation methods

- Meant to improve predictability and simplify timing analysis
- Time-triggered execution model (as opposed to Event-Driven)
  - Computations/Tasks remain inside statically-defined time reservations
- Enforced through mapping (allocation, scheduling)
  - Absence of interferences between cores
    - Two cores cannot access the same shared resource (e.g. a RAM bank) at the same time
    - Ensured by scheduling and resource (memory) allocation
  - Separate computations from communications
    - Globally: BSP (bulk synchronous parallel)
      - Alternating phases of computation (without communication) and global synchronization/communication
      - Often used along with memory allocation (e.g. one memory bank per core)

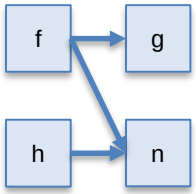
Time-triggered vs. Event-driven execution

- Use of TT where it's needed to enforce real-time requirements, ED elsewhere for robustness

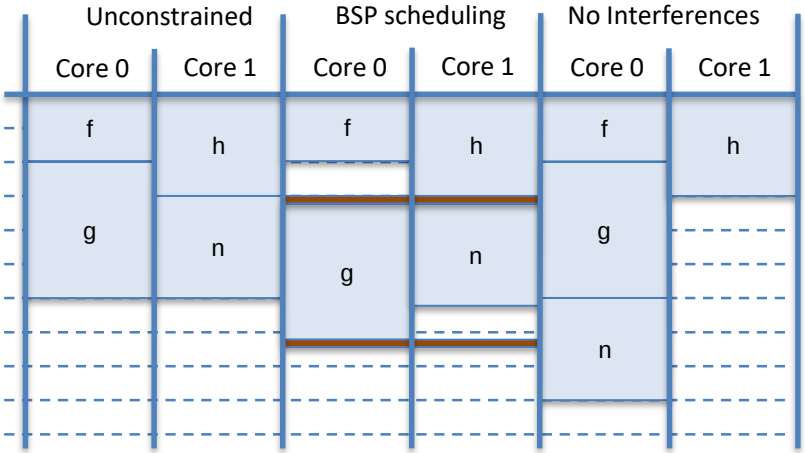


Scheduling-enforced properties

- Constraints reduce the solution space => efficiency loss
  - Intuition:



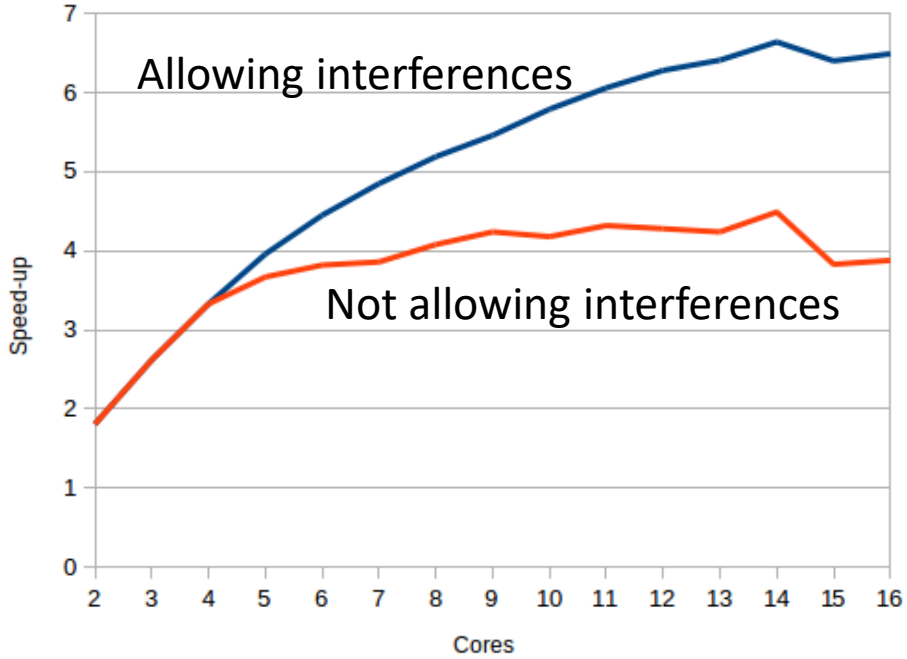
Functional specification



Three possible schedules

Scheduling-enforced properties

- Constraints reduce the solution space => efficiency loss
  - Flight controller application
    - No other isolation property
  - Significant penalty



Application (re-)structuring

- Parallelizing requires exposing potential parallelism (concurrency)
  - If your application is intrinsically sequential, parallelization does not help
  - Not exposing parallelism -> significant penalty
    - Automatic parallelization methods exist, but they add to implementation/certification cost
- Aircraft engine control:
  - Version 1: One large sub-system seen as a single, sequential task
    - Theoretical limit on parallelization speed-up: 1.8x (1.74x attained on 4 cores)
  - Version 2: Sub-system internal concurrency exposed (20% more nodes)
    - Theoretical limit on parallelization speed-up: 2.69x (2.66x attained on 4 cores)

Conclusion

- First evaluation of the cost of common isolation properties on large-scale use cases

- Time/Space isolation should be modulated depending (also) on performance needs
  - Subject to (strict) safety requirements
  - Trade-off with ease of development
  - Tools are here