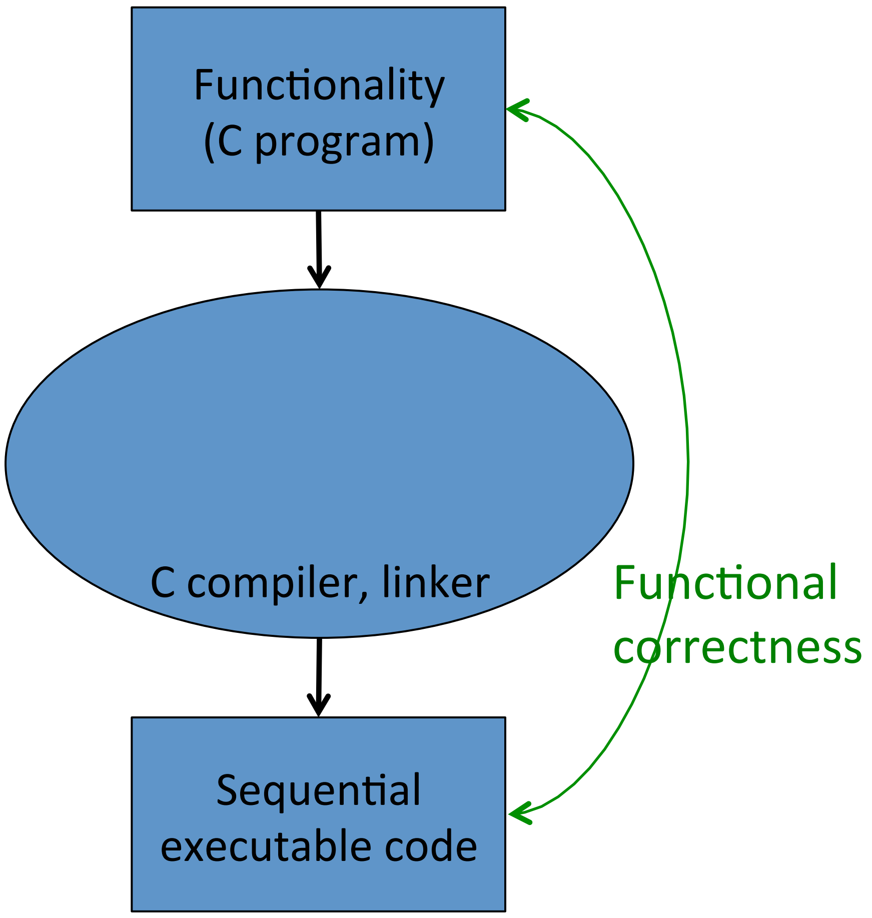
System-level compilation of parallel real-time systems

D. Potop-Butucaru

(et K. Didier, A. Cohen, G. Iooss, A. Graillat, J. Souyris, P. Baufreton)

Verimag Workshop – April 29, 2019

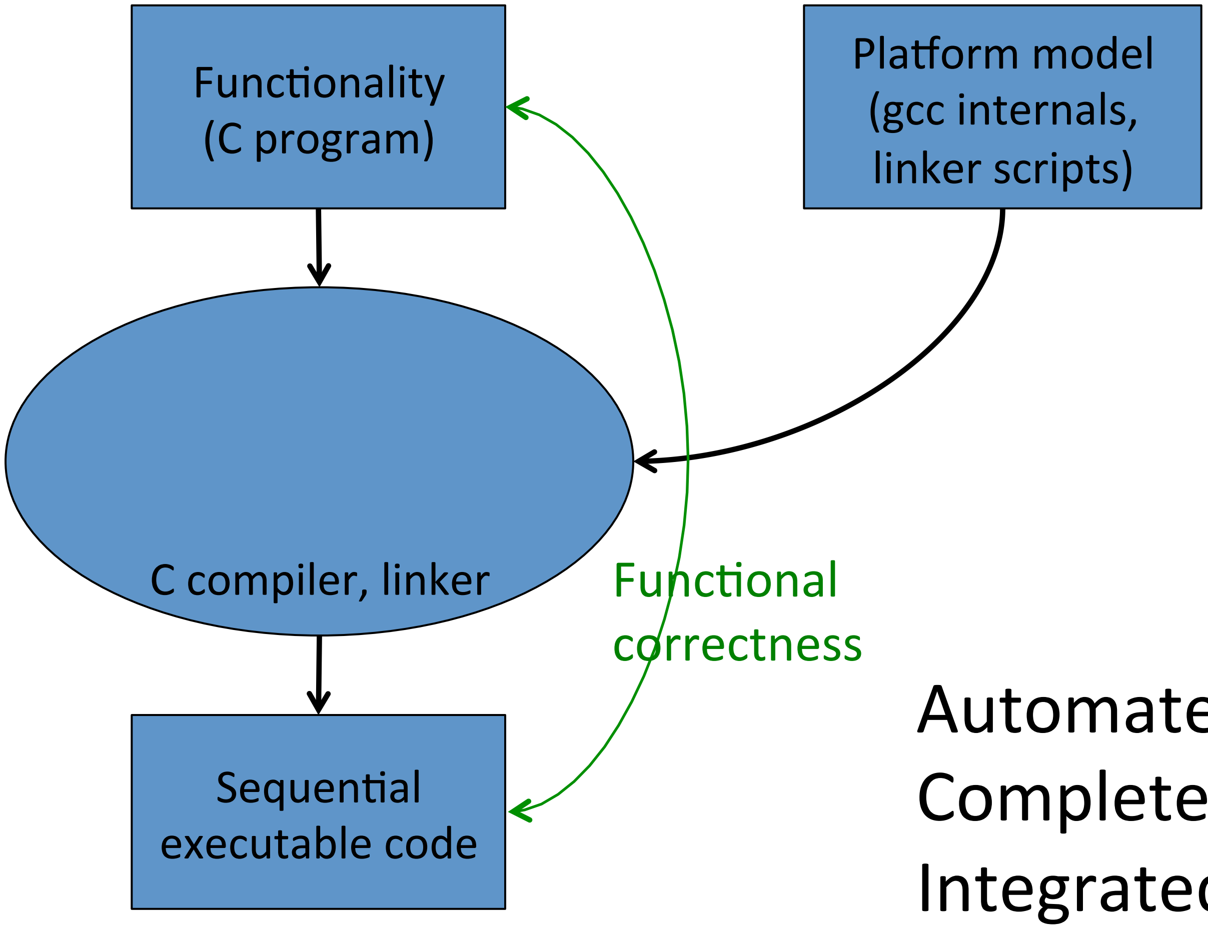
"High-level" language compilation



Automated
Complete
Integrated

C, Ada

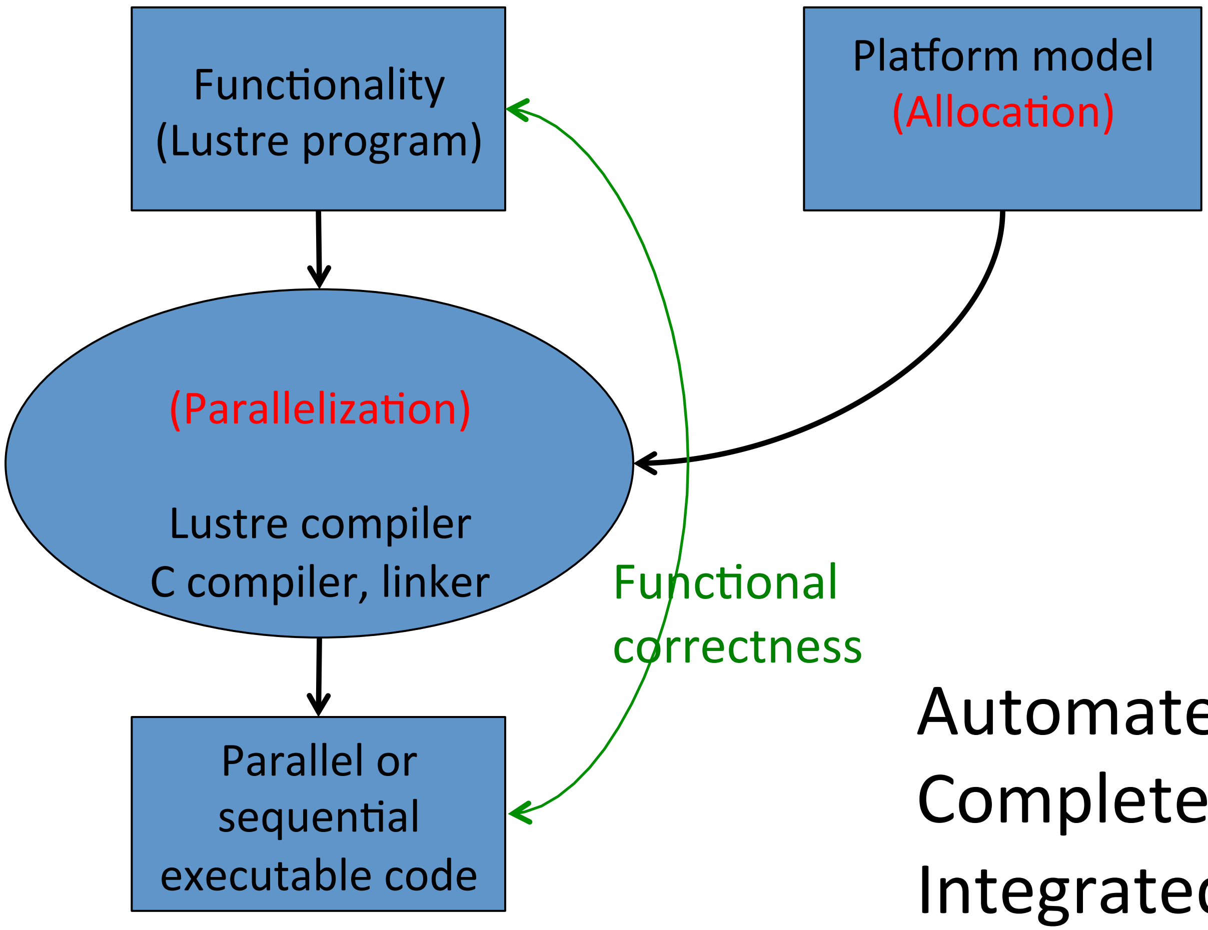
"High-level" language compilation



Automated
Complete
Integrated

C, Ada

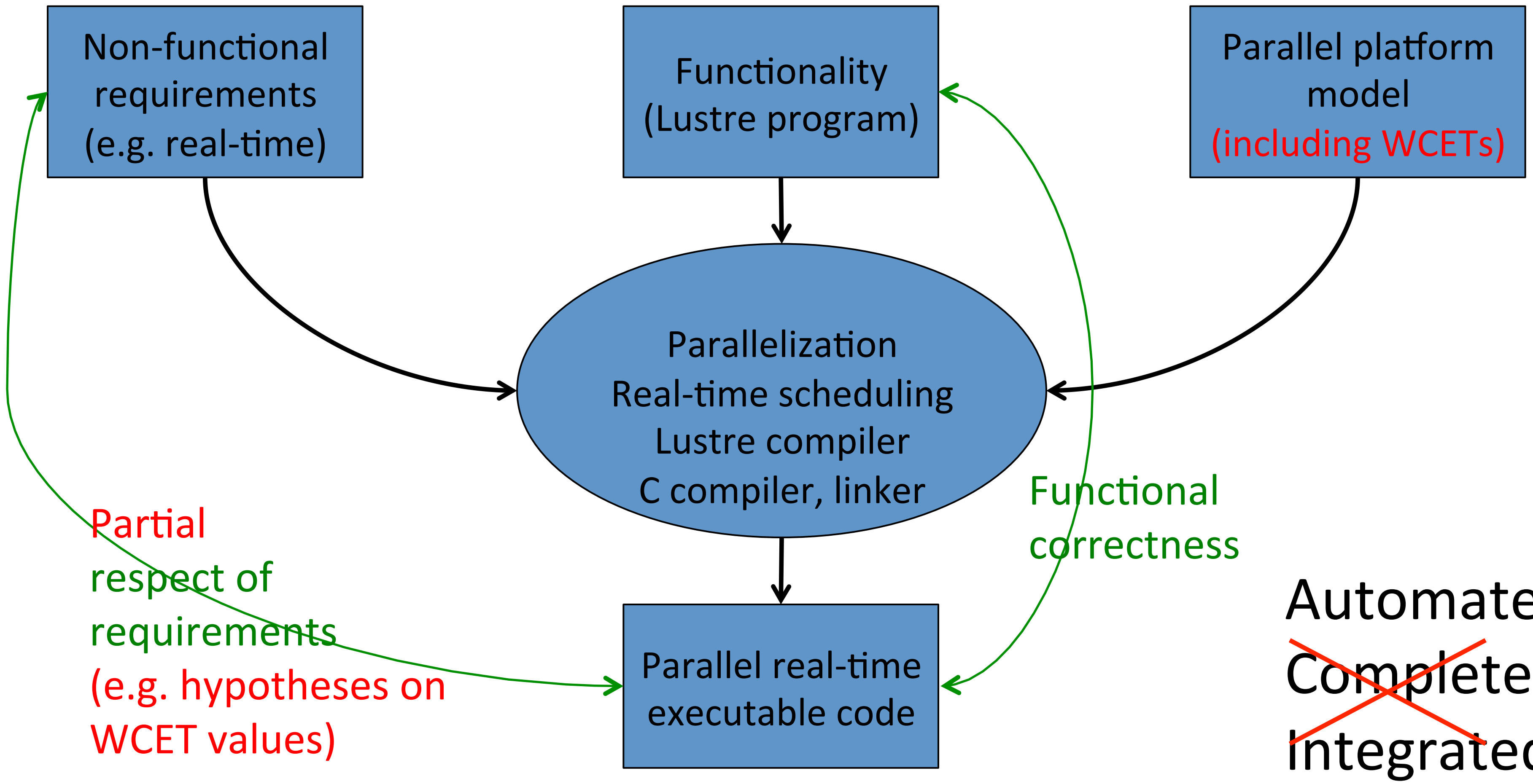
Data-flow (Lustre) compilation



Automated
Complete
Integrated

Lustre, Heptagon,
SCADE KCG Parallel (MCG)

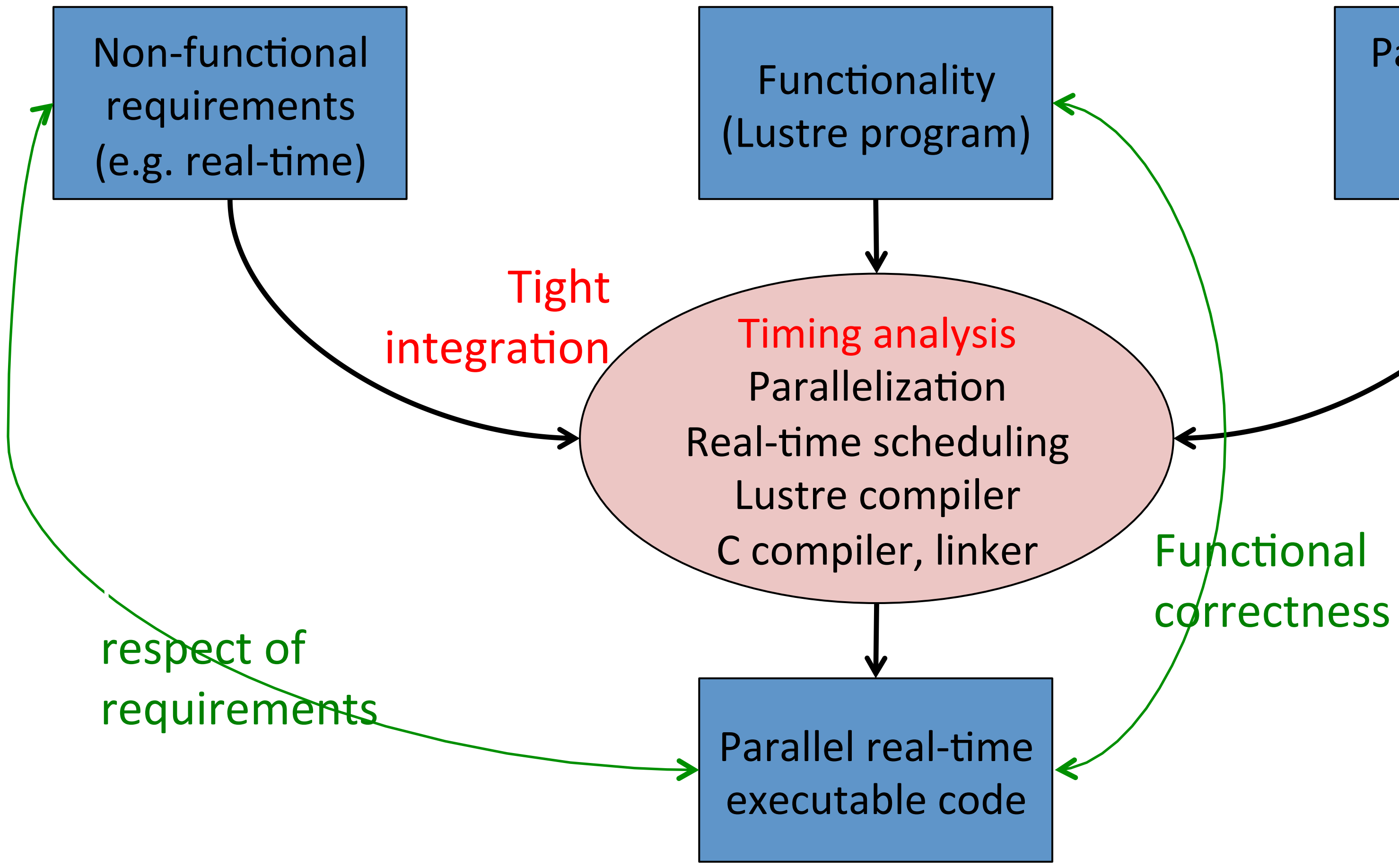
Real-time implementation flows


Non-functional requirements (e.g. real-time)
Functionality (Lustre program)
Parallel platform model (including WCETs)
Parallelization
Real-time scheduling
Lustre compiler
C compiler, linker
Parallel real-time executable code
Functional correctness
Partial respect of requirements (e.g. hypotheses on WCET values)

Automated
~~Complete~~
Integrated

SynDEx, Lustre2TTA,
Giotto, Lopht, Prelude, Asterios Developer, Simulink RT

Real-time systems compilation



Automated
Complete
Integrated

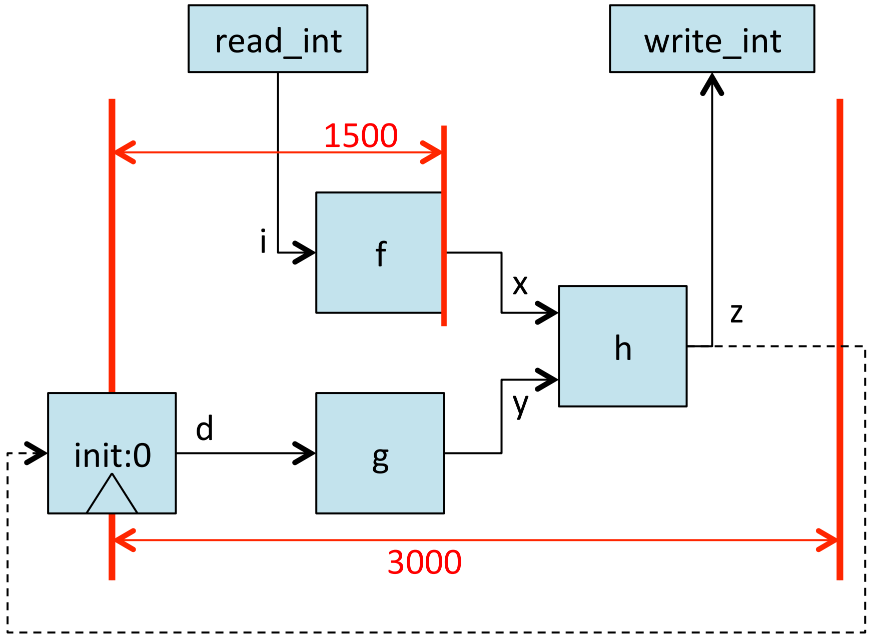
Previous work assuming no interferences (HW or SW isolation)

New Lopht: allows interferences for efficiency

Platform-independent specification

- Cyclic, deterministic, non-func. requirements

```
open Io (* platform-dependent I/O functions read_int, write_int *)
open Func (* externally-defined functions f, g, h *)
node simple (i:int) returns (o:int)
var y,d:int; x:float;
let
  deadline(1500) x = f(i);
                 y = g(d);
                 o = h(x,y);
                 d = 0 fby o;
tel

period(3000) node main () returns ()
var i,o : int;
let
   i = read_int();
   o = inlined simple (i);
  () = write_int(o);
tel
```



Architecture description

```
Architecture

Cores:2

Memory Bank Number : 16
Bank Size : 0x20000

Memory Excluded
[Start:0x000000 End:0x060000]
[Start:0x1ff000 End:0x200000]
```

- One compute cluster
  - Focus on fine-grain, efficient parallelization in shared memory
  - NoC scheduling in previous work (Carle et al., 2015)
- Architecture description
  - Resources to use on the compute tile
  - Number of:
    - cores to use
    - memory banks to use
      - Size is also provided, to allow for evaluation and easy extension
  - Memory areas that cannot be used

Real-time parallelization

```
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();
   }
}
```

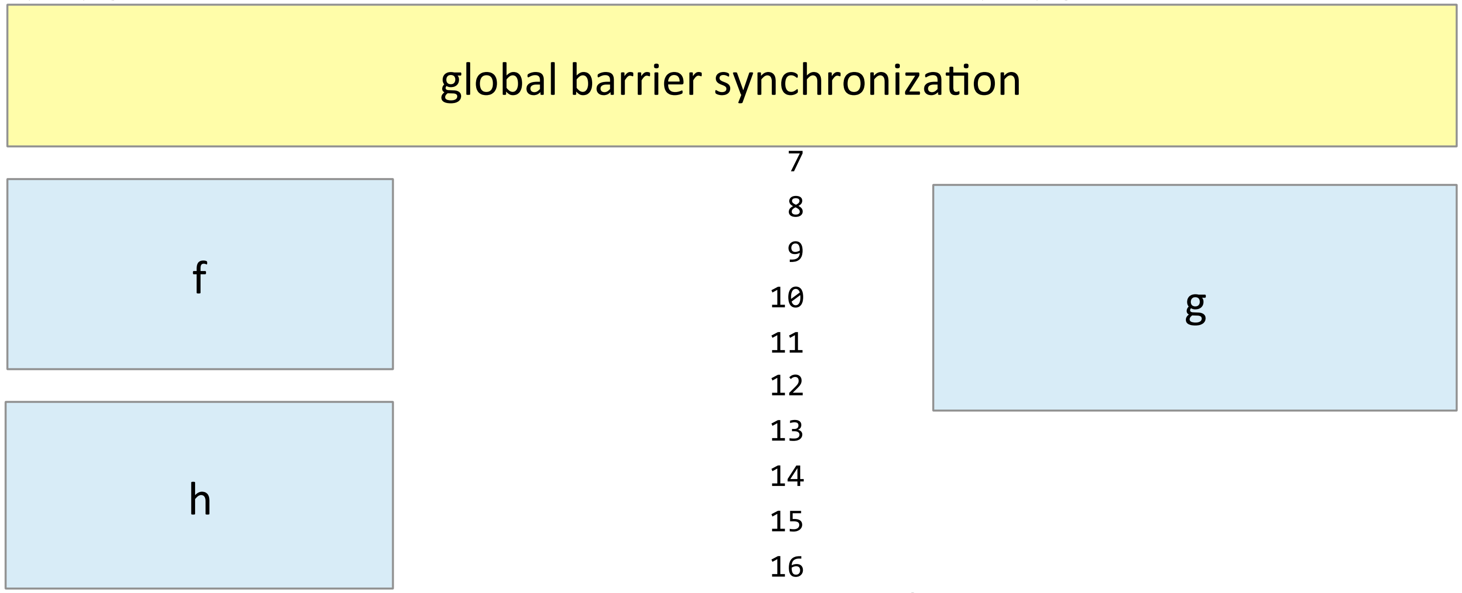
```
void* thread_cpu1(void* unused){
   lock_init_pe(1);
   for(;;){


      global_barrier_sync(1);

      dcache_inval();
      g(z,&y);
      dcache_flush();
      lock(1,1);
      unlock(0);




   }
}
```

- Multi-threaded implementation (non-preemptive)
  - Fully static memory allocation
  - Static hard real-time guarantees

Real-time parallelization

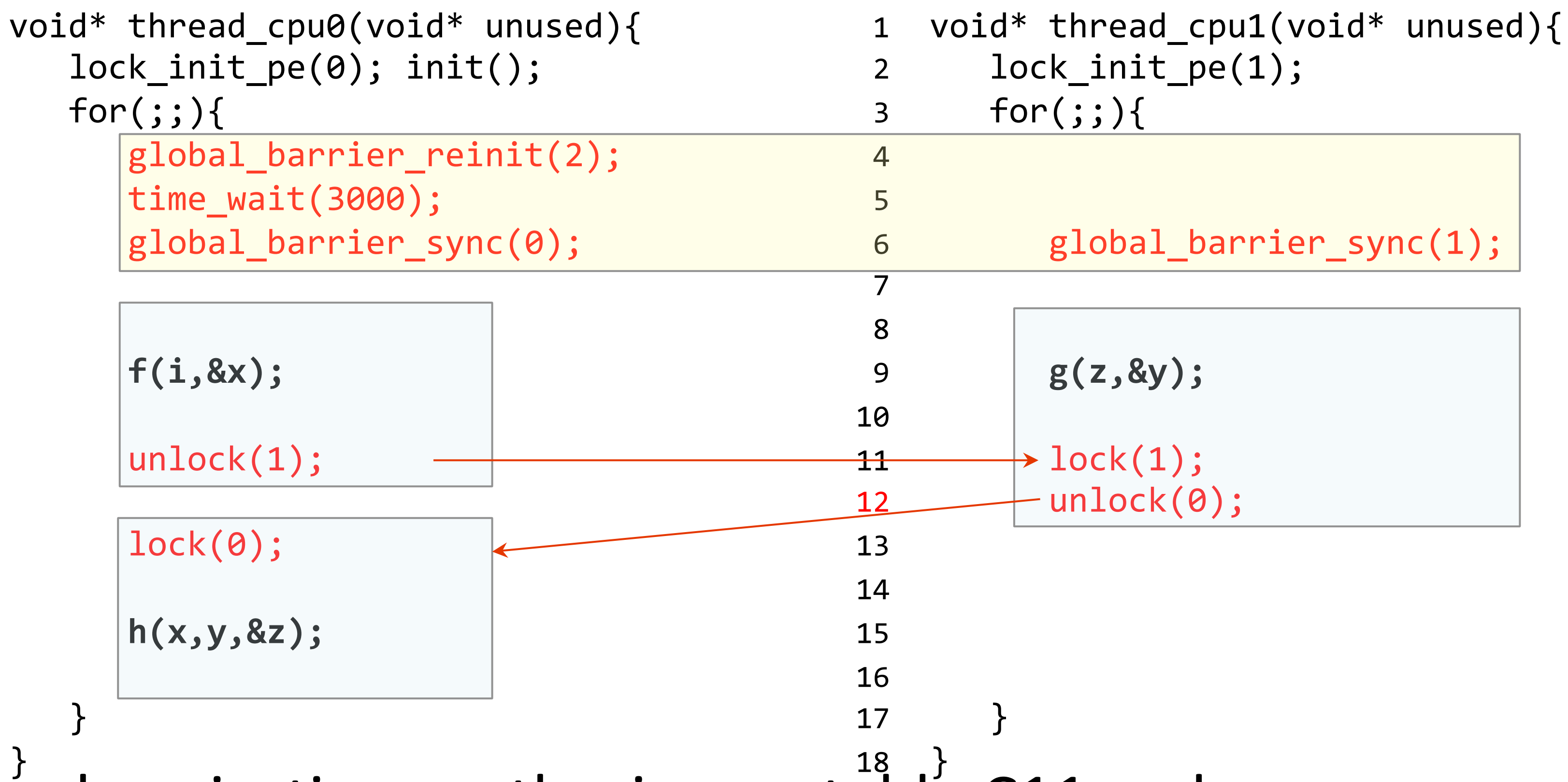
```
void* thread_cpu0(void* unused){
   lock_init_pe(0); init();
   for(;;){
   }
}
```

```
void* thread_cpu1(void* unused){
   lock_init_pe(1);
   for(;;){
   }
}
```



- Parallelization of dataflow blocks
  - Static real-time scheduling (timetabling)
  - Attention: worst-case time reservations
    - Sequential WCET analysis precision (e.g. aiT, OTAWA) is a limit

Real-time parallelization

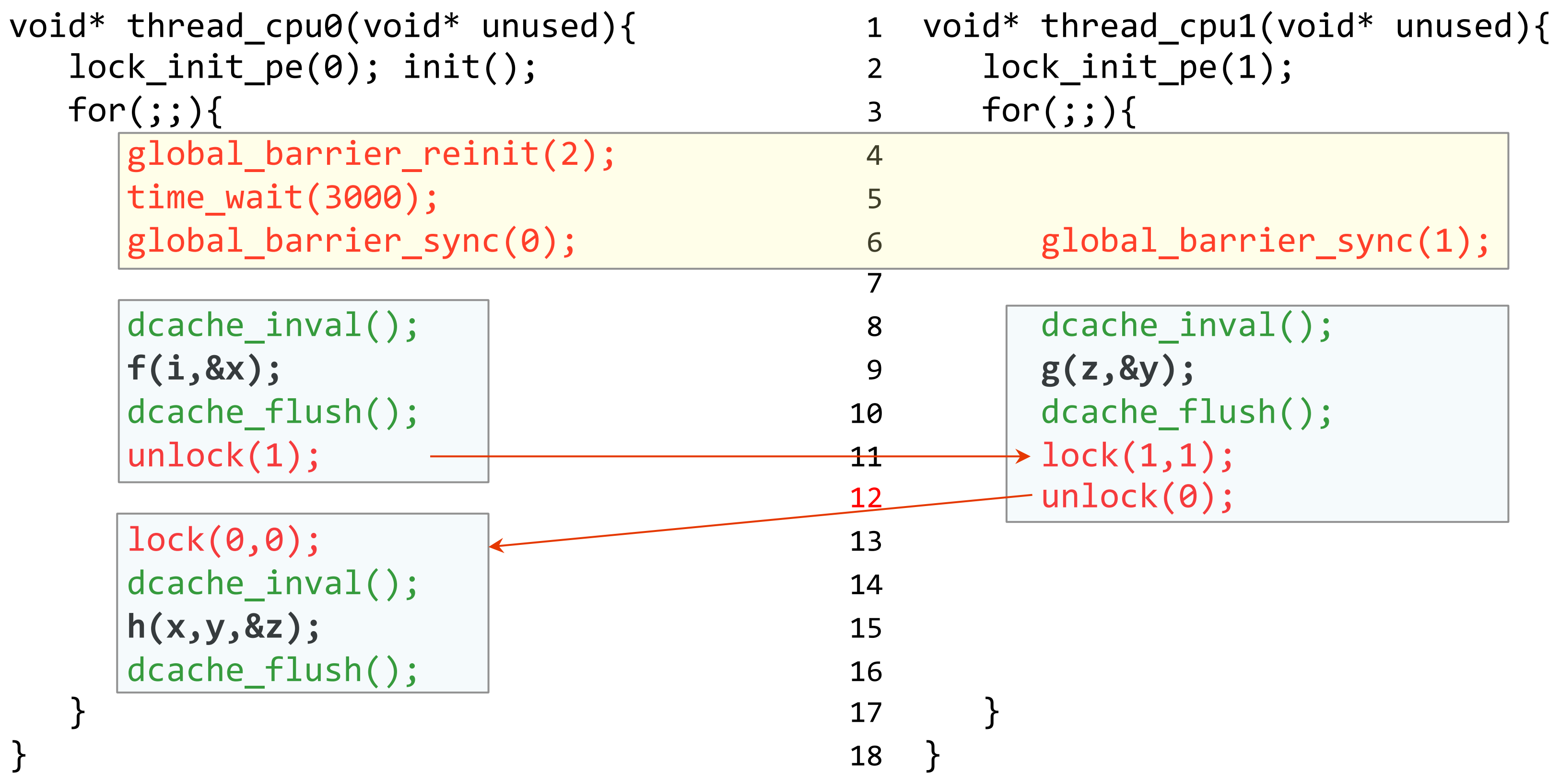


```
void* thread_cpu0(void* unused){
   lock_init_pe(0); init();
   for(;;){
      global_barrier_reinit(2);
      time_wait(3000);
      global_barrier_sync(0);


      f(i,&x);

      unlock(1);

      lock(0);

      h(x,y,&z);

   }
}
```

```
void* thread_cpu1(void* unused){
   lock_init_pe(1);
   for(;;){


      global_barrier_sync(1);


      g(z,&y);

      lock(1);
      unlock(0);





   }
}
```

- Synchronization synthesis : portable C11 code
  - sub-set of C11 concurrency model + time_wait
  - functional determinism (and data race free, thread-safe)
  - bounded per-node synchronization code (very important)

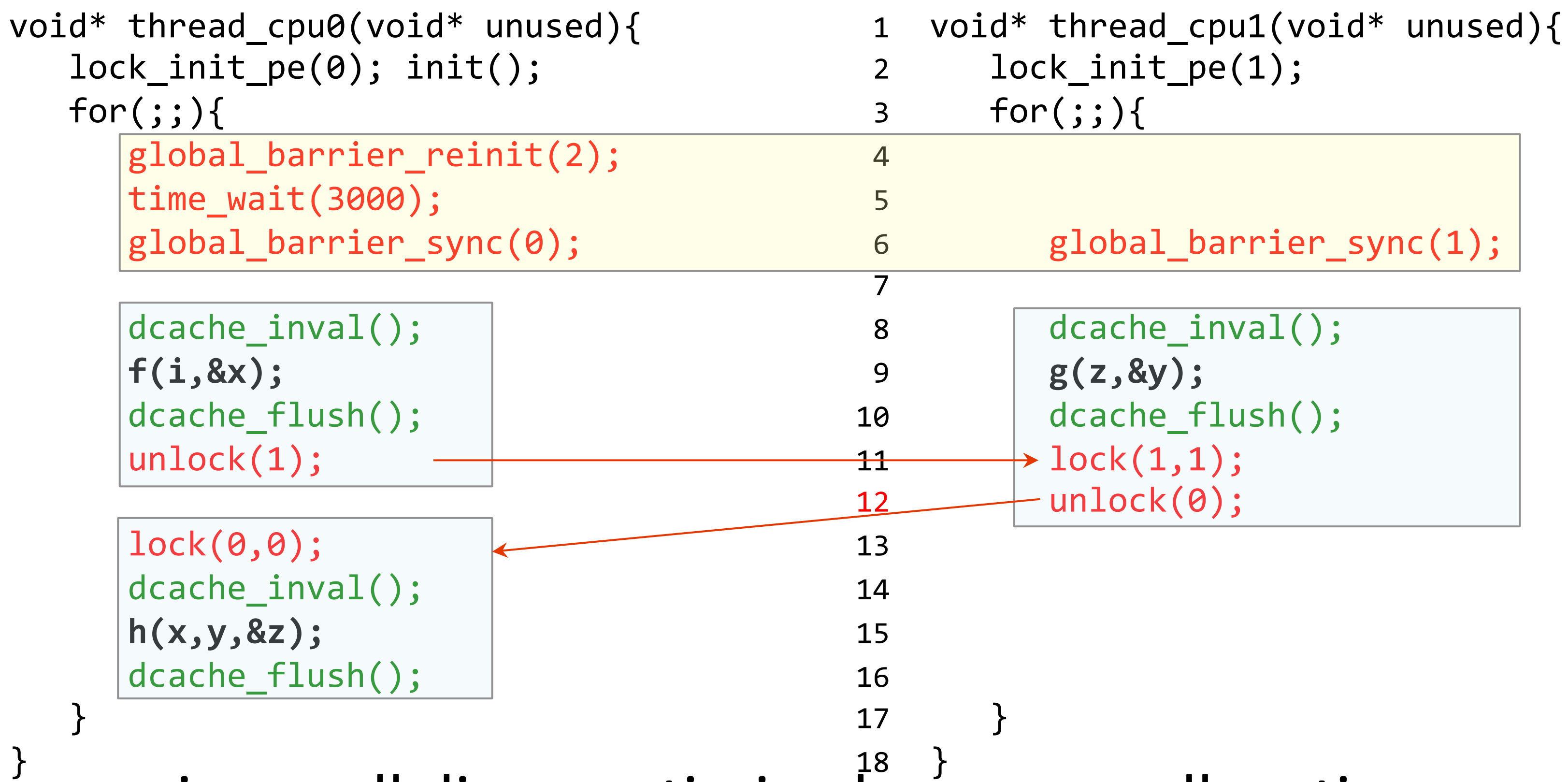
Real-time parallelization



```
void* thread_cpu0(void* unused){            1   void* thread_cpu1(void* unused){
   lock_init_pe(0); init();                 2      lock_init_pe(1);
   for(;;){                                 3      for(;;){
      global_barrier_reinit(2);             4
      time_wait(3000);                      5
      global_barrier_sync(0);               6         global_barrier_sync(1);
                                            7
      dcache_inval();                       8         dcache_inval();
      f(i,&x);                              9         g(z,&y);
      dcache_flush();                      10         dcache_flush();
      unlock(1);                           11         lock(1,1);
                                           12         unlock(0);
      lock(0,0);                           13
      dcache_inval();                      14
      h(x,y,&z);                           15
      dcache_flush();                      16
   }                                       17      }
}                                          18   }
```

- Cache coherency synthesis
  - Mandatory on Kalray MPPA256 Bostan

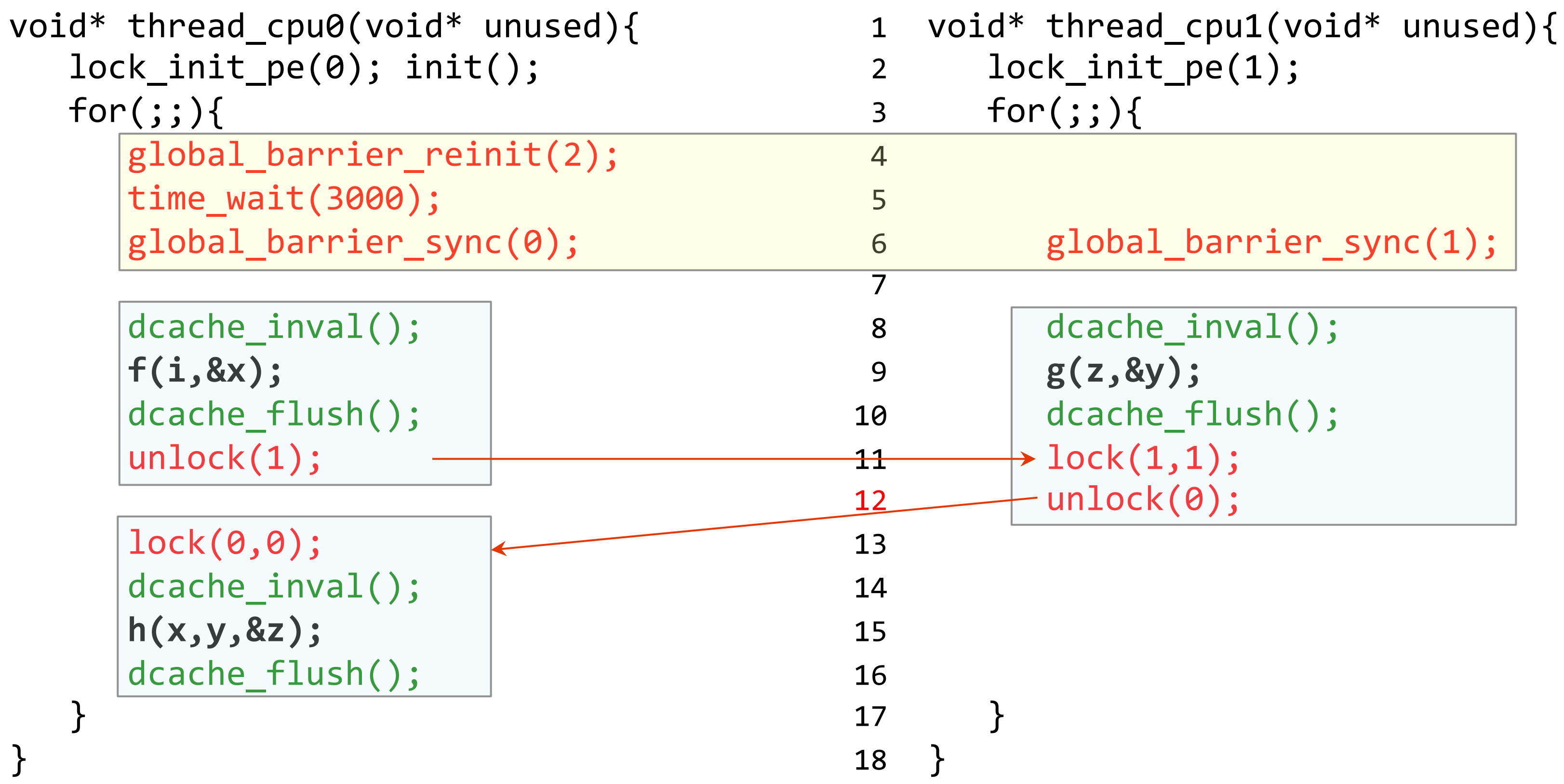
Real-time parallelization



```
void* thread_cpu0(void* unused){          void* thread_cpu1(void* unused){
    lock_init_pe(0); init();                  lock_init_pe(1);
    for(;;){                                  for(;;){
        global_barrier_reinit(2);
        time_wait(3000);
        global_barrier_sync(0);                   global_barrier_sync(1);

        dcache_inval();                           dcache_inval();
        f(i,&x);                                  g(z,&y);
        dcache_flush();                           dcache_flush();
        unlock(1);                                lock(1,1);
                                                  unlock(0);
        lock(0,0);
        dcache_inval();
        h(x,y,&z);
        dcache_flush();
    }                                         }
}                                         }
```

- Fine-grain parallelism, optimized resource allocation
  - Take advantage of low-cost sync, communication
    - No per-CPU variable copies, when space isolation is not required
    - No copy operations

Real-time parallelization



```
void* thread_cpu0(void* unused){          void* thread_cpu1(void* unused){
   lock_init_pe(0); init();                  lock_init_pe(1);
   for(;;){                                  for(;;){
      global_barrier_reinit(2);
      time_wait(3000);
      global_barrier_sync(0);                   global_barrier_sync(1);

      dcache_inval();                           dcache_inval();
      f(i,&x);                                  g(z,&y);
      dcache_flush();                           dcache_flush();
      unlock(1);                                lock(1,1);
                                                unlock(0);
      lock(0,0);
      dcache_inval();
      h(x,y,&z);
      dcache_flush();
   }                                         }
}                                         }
```

- Very efficient data variable allocation
  - Can do a lot better (70% reduction) if observability requirements are relaxed

Memory allocation

```
. = thr0_ALLOC ;
.text_thread0 ALIGN(64) : {
  thread_cpu0.o(.text)
}
. = data_thread0 ALIGN(32) : {
  thread_cpu0.o(.data)
  thread_cpu0.o(.bss)
  thread_cpu0.o(.rodata)
}
. = thr0_STACK ;
_user_stack_start0 = .;
```

- Code placement entirely controlled
  - Threads
    - Code and local data contiguously at start of the bank
    - Stack at the end of the bank

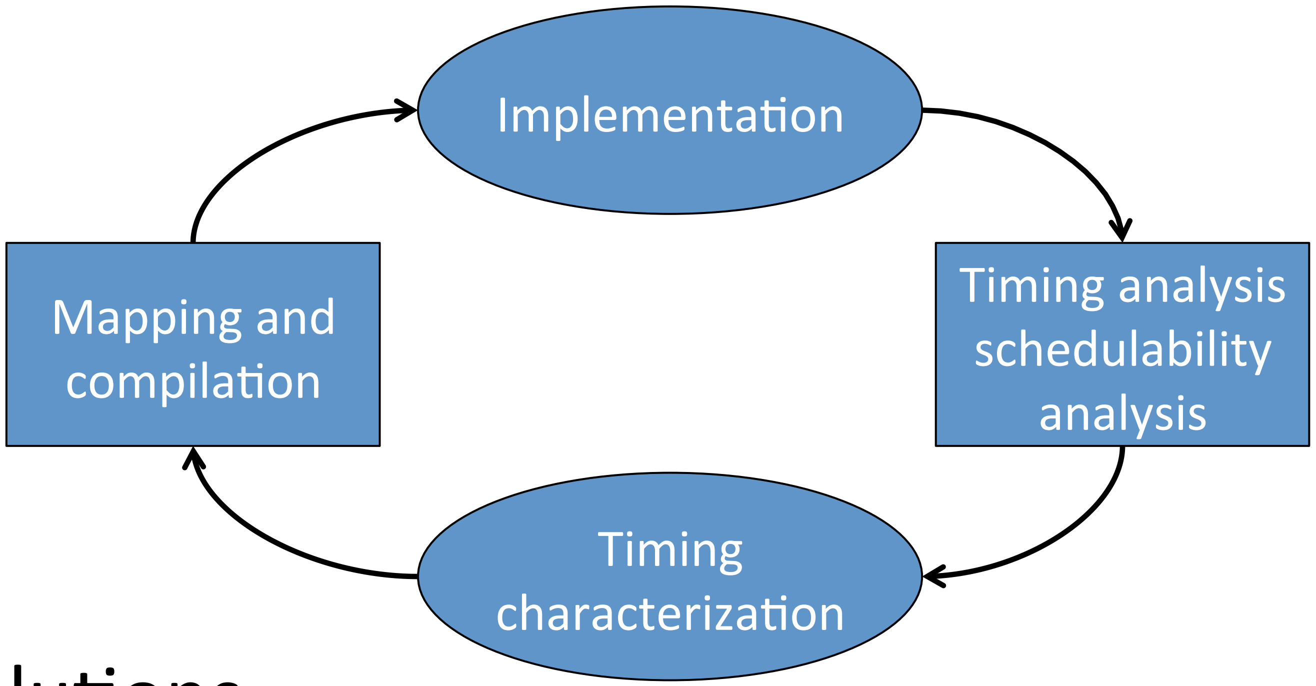
```
. = f_ALLOC ;
.f_text ALIGN(ICACHE_LINE_SIZE) : {
  f.o(.text)
}
.f_data ALIGN(DCACHE_LINE_SIZE) : {
  f.o(.data)
  f.o(.bss)
  f.o(.rodata)
}
```

  - Nodes
    - Code and local data contiguously

```
x = x_ALLOC;
```

  - Data-flow variables placed in the remaining space

The real-time mapping problem



- Solutions
  - Implement using unsafe characteristics, then determine if implementation satisfies requirements
  - Use over-approximated timing characterization that cover all possible mappings

The real-time mapping problem

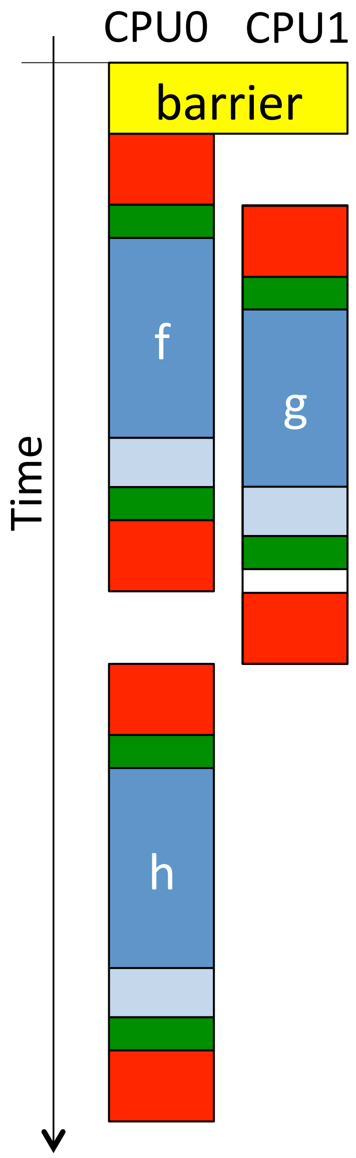
- Solutions
  - Implement using unsafe characteristics, then determine if implementation satisfies requirements
    - Choosing unsafe characteristics may be difficult
      - Too small values → non-sched. ; too large → resource waste
    - Iterate the process: how to ensure convergence?
  - Use over-approximated timing characterization that cover all possible mappings
    - Produces a safe implementation
      - Our choice (long tradition of AAA)
    - Over-approximation costs
      - Need precise timing models for efficient resource allocation (like the one we considered) → analysis is more expensive

Mapping heuristic

- Compilation-like heuristics
  - ability to include accounting for interferences
  - scalability
  - complex code transformations
- List scheduling
  - When considering a node:
    - Allocate its code and yet unallocated data to memory
    - Allocate its execution to one of the processing cores
      - Timetable
    - Choose its start date
      - Constraints: data dependencies, real-time requirements
      - Ensure that previously-mapped nodes still respect requirements

Mapping heuristic

- Reserved(f) = WCET(f) + overheads(f)
- Need low worst-case bounds on
  - Coherency costs (easy on Kalray)
  - Synchronization costs
    - HW support -> low operation overhead
    - Novel synchronization protocol
  - Interferences
    - **Including by not-yet-scheduled nodes**



Interferences

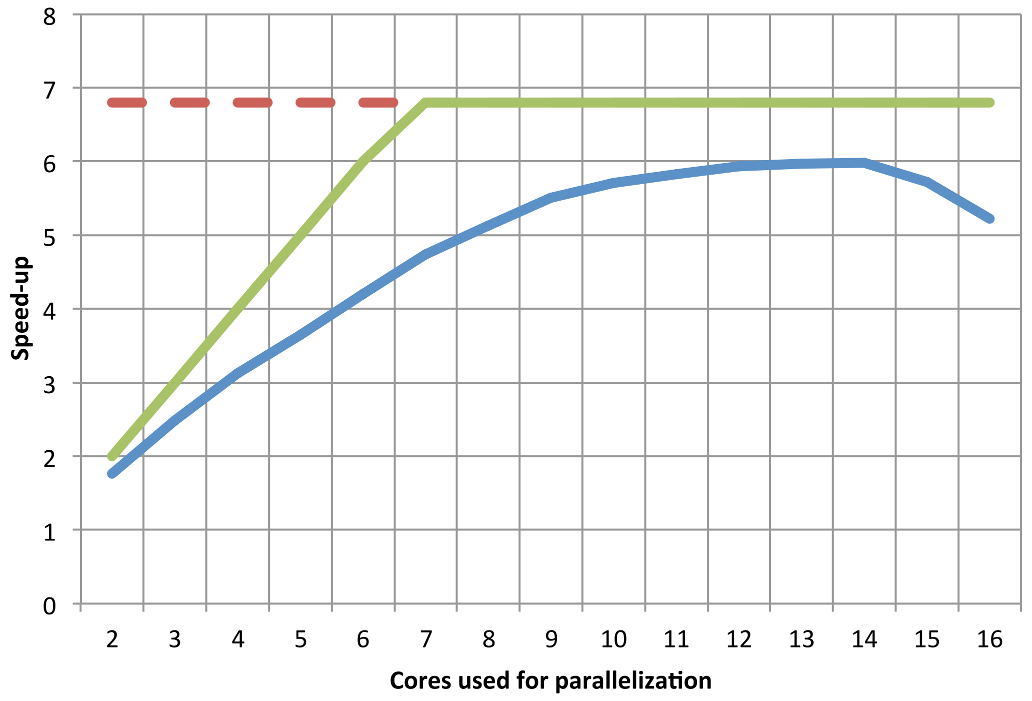
- Need to provision acceptable interferences before scheduling
  - Bound on interferences by not yet scheduled functions
- Increase each WCET by a percentage of interference provisions
  - Lopht compiler parameter
  - Typical optimal values: 0% (on 2 cores) 10% (on 8 cores)
- When mapping a function, check that its interferences and those of all already mapped functions remain lower than provisioned
  - If not, search for a later date
  - Percentage = 0% => no interferences (old Lopht [Carle at al.2012])
    - Low parallelization on Kalray MPPA (may work on other architectures)
  - **Choosing the right value is important for efficiency**
    - **Too low → no sharing, too high → over-provisioning**

Experimental results

- Evaluation:
  - Functional correctness
    - Trace comparison w.r.t. sequential
  - Scalability of the tools
    - Speed of compilation and code generation
  - Speed efficiency of the generated code
    - Reduce period w.r.t. sequential case
    - Worst-case guarantees vs. measured performance
- Full results in Inria RR-9180, to appear in ACM TACO

UC1

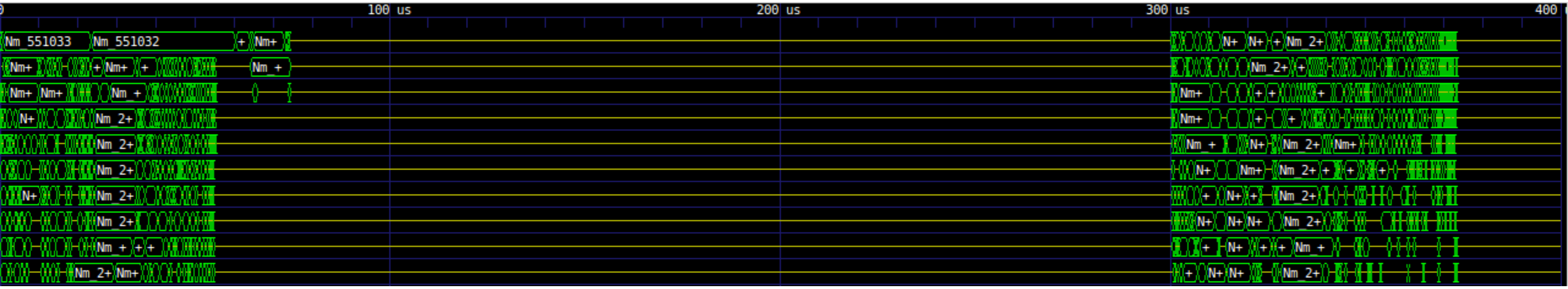
- ~5k unique nodes, ~36k unique dataflow vars
- Multi-period
  - ~18k instances
- Scalability
  - 8 cores: ~22s
  - 16 cores: ~43s



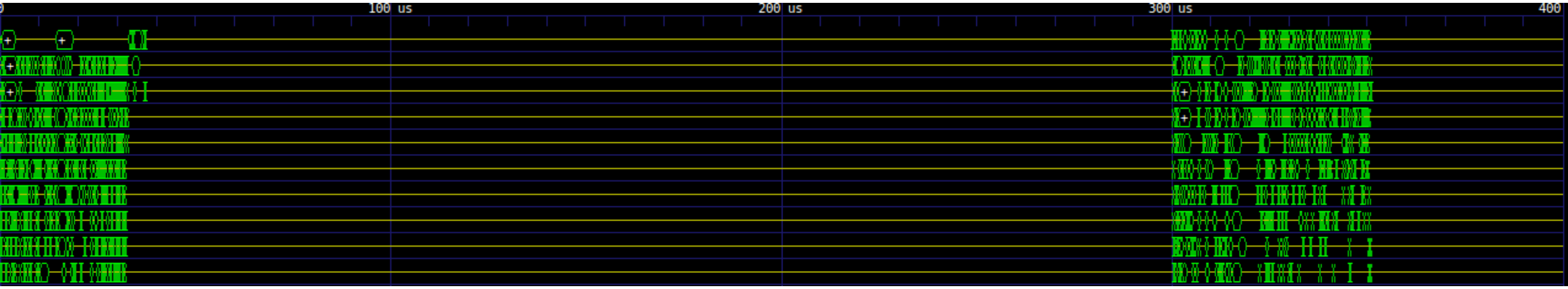
- Parallelization
  - CP limit: 6.8x
  - 2 cores->1.76x, 4 cores->3.12x, 8 cores->5.13x, 12 cores->5.93x
  - Bandwidth saturation beyond 12 cores

UC1

- Node execution
  - Schedule



  - Actual execution



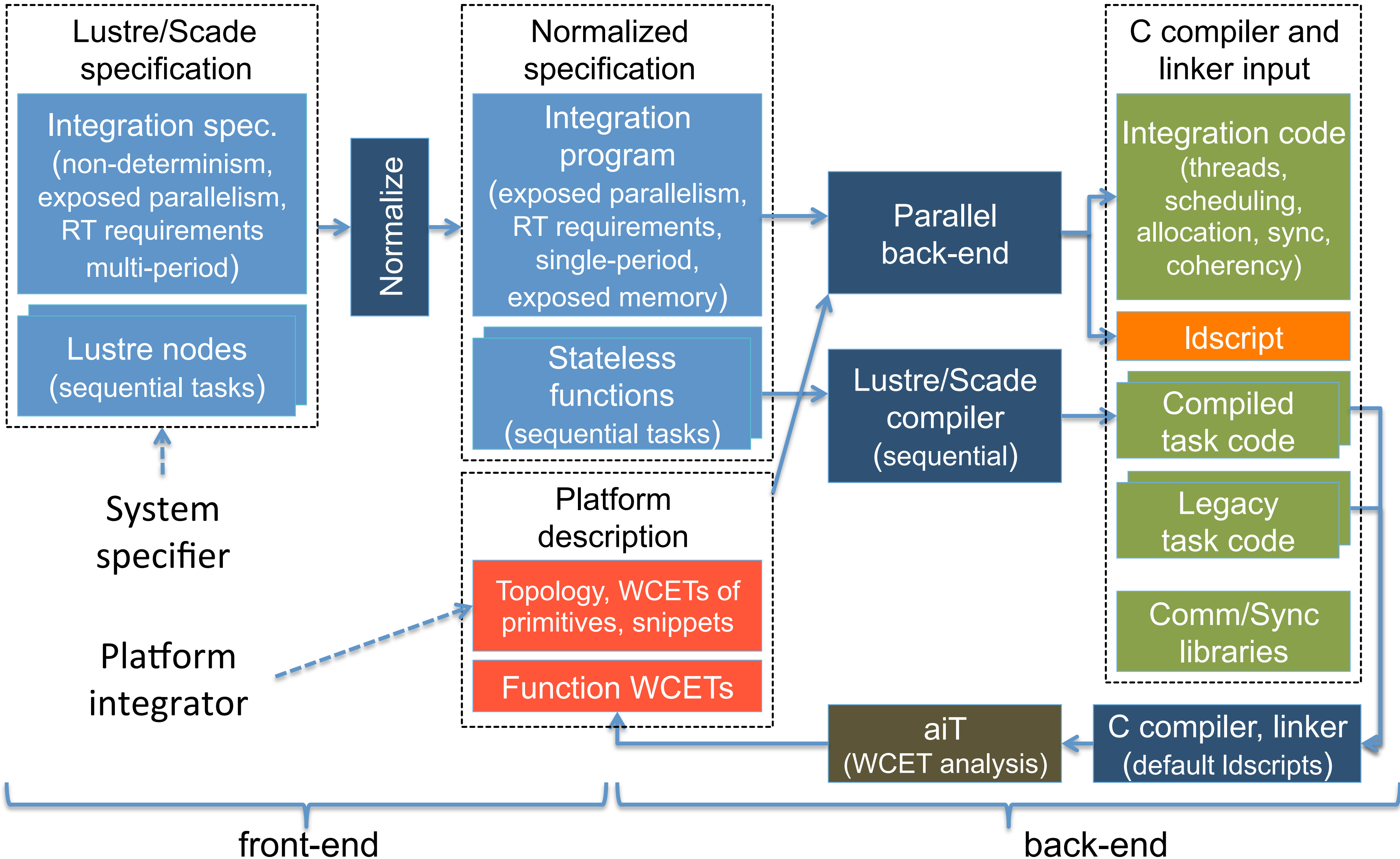
Conclusion

- Real-time systems compiler
  - End-to-end full automation
    - Scalable
  - Correct by construction
    - Including respect of non-functional requirements
    - No need for post-scheduling timing analysis
      - No convergence problem
  - Efficient
    - Memory allocation
    - Synchronization
    - Good practical results for two industrial case studies
  - Fine-grain parallelization
    - Resource sharing (when isolation is not required) is important!

Ongoing and future work

- Other platforms
  - Kalray Coolidge
  - Full Kalray MPPA256 chip
    - Code and data overlays and scheduling over NoC
  - ARMv8, T1042, Infineon multi-cores
    - What guarantees can be preserved?
- Trade-offs between generated code size, speed, and isolation properties
- Correctness formalization and formal validation
  - First results in ACSD'19

Compilation flow



Lustre/Scade specification
Integration spec. (non-determinism, exposed parallelism, RT requirements multi-period)
Lustre nodes (sequential tasks)
Normalize
Normalized specification
Integration program (exposed parallelism, RT requirements, single-period, exposed memory)
Stateless functions (sequential tasks)
Parallel back-end
Lustre/Scade compiler (sequential)
C compiler and linker input
Integration code (threads, scheduling, allocation, sync, coherency)
ldscript
Compiled task code
Legacy task code
Comm/Sync libraries
System specifier
Platform integrator
Platform description
Topology, WCETs of primitives, snippets
Function WCETs
aiT (WCET analysis)
C compiler, linker (default ldscripts)
front-end
back-end