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

Functionality (C program) → C compiler, linker → Sequential executable code → Functional correctness

C, Ada

Automated Complete Integrated
"High-level" language compilation

Functionality (C program) → C compiler, linker → Sequential executable code → Platform model (gcc internals, linker scripts)

Automated Complete Integrated

C, Ada
Data-flow (Lustre) compilation

- Functionality (Lustre program)
- (Parallelization)
- Lustre compiler
- C compiler, linker
- Parallel or sequential executable code
- Platform model (Allocation)

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

Partial respect of requirements (e.g. hypotheses on WCET values)

Functional correctness

Automated Complete
Integrated

SynDEx, Lustre2TTA, Giotto, Lopht, Prelude, Asterios Developer, Simulink RT
Real-time systems compilation

Non-functional requirements (e.g. real-time)

Tight integration

Functionality (Lustre program)

Parallel platform model (no WCETs)

Timing analysis
Parallelization
Real-time scheduling
Lustre compiler
C compiler, linker

Parallel real-time executable code

Respect of requirements

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

- 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

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

- 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

```c
void* thread_cpu0(void* unused)
{
  lock_init_pe(0); init();
  for(;;){
    global_barrier_sync(1);
    dcache_inval();
    g(z,&y);
    dcache_flush();
    lock_request(1,1);
    lock_grant(0);
  }
}

void* thread_cpu1(void* unused)
{
  lock_init_pe(1);
  for(;;){
    global_barrier_reinit(2);
    time_wait(3000);
    global_barrier_sync(0);
    dcache_inval();
    f(i,&x);
    dcache_flush();
    lock_grant(1);
    lock_request(0,0);
    dcache_inval();
    h(x,y,&z);
    dcache_flush();
  }
}
```
Real-time parallelization

- 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

- Cache coherency synthesis
  - Mandatory on Kalray MPPA256 Bostan
Real-time parallelization

- 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

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

- Code placement entirely controlled
  - Threads
    - Code and local data contiguously at start of the bank
    - Stack at the end of the bank
  - Nodes
    - Code and local data contiguously
  - Data-flow variables placed in the remaining space

```asm
.. = 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 = ..;

.. = 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)
}

x = x_ALLOC;
```
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 $\rightarrow$ non-sched.; too large $\rightarrow$ 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) $\rightarrow$ 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

Front-end

System specifier

- Lustre/Scade specification
  - Integration spec. (non-determinism, exposed parallelism, RT requirements multi-period)
  - Lustre nodes (sequential tasks)

- Platform specifier

- Platform integrator

Back-end

- C compiler and linker input
  - Integration code (threads, scheduling, allocation, sync, coherency)
  - Lustre/Scade compiler (sequential)

- aiT (WCET analysis)
  - Compiled task code
  - Legacy task code
  - Comm/Sync libraries

- C compiler, linker (default ldscripts)

Integration spec. program (exposed parallelism, RT requirements, single-period, exposed memory)

Integration code (threads, scheduling, allocation, sync, coherency)

Stateless functions (sequential tasks)

Platform description

- Topology, WCETs of primitives, snippets
- Function WCETs

Normalized specification

- Integration program

Parallel back-end

Compiled libraries

- Normalization

Normalized spec. (c)

Compiled task code