A compilation-like approach to real-time systems implementation

Keryan Didier
INRIA, AOSTE2 team
12/4/17
The need for automation

- Higher level program specification
- Implementation automation

Engineer struggling to keep her code from crashing, 1969
Compilation of « high-level » languages

- Functionality (C program)
- Compiler Linker
- Sequential executable code
Compilation of « high-level » languages

- Functionality (C program)
- Compiler Linker
- Sequential executable code

Platform model (gcc internals, linker scripts)
Data-flow (Lustre) compilation

Functionality (Lustre program)

(Parallelization)
Lustre compiler
C compiler
Linker

Platform model, Allocation

Parallel or sequential executable code
Real-time data-flow compilation

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

Functionality (Lustre program)

Parallel platform model

Parallelisation
Real-time scheduling
Lustre compiler
C compiler
Linker

Parallel real-time executable code
Related work (1/2)

• « Classical » compilation
  – Back-end and optimization
    • Software pipelining
    • Scheduling on VLIW architectures with exposed pipelines
  – Precise timing models to achieve efficiency
    • Timing of basic operations does not depend on allocation and scheduling
  – Average-case optimization (vs. worst-case satisfaction)

• Off-line and time-triggered real-time scheduling
  – SynDEx, Lustre2TTA, Giotto, Prelude, Lopht, Asterios Developer, etc.
  – Front-end: Significant front-end work (we do not insist on it here)
  – Back-end: Existing tools assume the existence of a timing characterization satisfying some properties
    • How to derive it?
    • What is the cost of mapping choices and generated code?
Related work (2/2)

- Parallel, possibly real-time code generation without schedulability guarantees
  - Simulink Real-Time, SCADE KCG6 parallel
- Automatic parallelization, parallel compilation
- WCET analysis of parallel code
  - Heptane, OTAWA
Outline

• Input: Data-flow programming in Lustre
  – And timing extensions
• Output: Structure of an implementation
• Timing model
• Resource allocation and code generation
  – Compilation-like
• Experimental results
• Conclusion
node main () returns ()
var
  i : int; x : float;
  y : int; z : int;
  d : int;
let
  i = read_int();
  x = f(i);
  y = g(d);
  z = h(x, y);
  d = 0 fby z;
() = write_int(z);
tel
Data-flow programming in Lustre

```plaintext
node main () returns ()

var
  i : int; x : float;
  y : int; z : int;
  d : int;

let
  i = read_int();
  x = f(i);
  y = g(d);
  z = h(x, y);
  d = 0 fby z;
  () = write_int(z);

tel
```

- Data-flow in textual form
  - Cyclic execution
  - State elements: fby
Data-flow programming in Lustre

```plaintext
node main () returns ()
var
  i : int; x : float;
  y : int; z : int;
  d : int;
let
  i = read_int();
  x = f(i);
  y = g(d);
  z = h(x,y);
  d = 0 fby z;
  () = write_int(z);
tel
```

- Simple, deterministic concurrency
  - Static Single Assignment form
    - Each variable assigned exactly once
  - Functions f, g, and h specified externally in C or Lustre
    - No recursion, no side-effects, no heap

- Well understood semantics, analysis, compilation

- Integration specifications
  - System-level, no further composition
    - No input or output arguments
    - I/O done through specific functions (e.g. read/write of memory-mapped devices)
Non-functional requirements

- Real-time requirements
  - Period
  - Release dates
  - Deadlines
- Time unit: ms, µs, CPU cycle
- Other requirements
  - Allocation constraints
Structure of an implementation

- Multi-threaded C code
  - Initialization
  - Function calls
  - Synchronization
    - Between threads
    - With real time
  - Memory coherency

- Allocation of all code and data
  - Node code, thread code, stacks, data-flow variables
  - Linker scripts
Multi-threaded C code

```c
void* thread_cpu0(void* unused){
    lock_init_pe(0); init(); time_init(&time);
    for(;;){
        global_barrier_reinit(2);
        time+=3000; wait(time);
        global_barrier_sync(0);
        dcache_inval();
        f(i,&x);
        dcache_flush();
        lock_grant(1);
        lock_request(0,0);
        dcache_inval();
        h(x,y,&z);
    }
}

void* thread_cpu1(void* unused){
    lock_init_pe(1);
    for(;;){
        global_barrier_sync(1);
        dcache_inval();
        g(z,&y);
        dcache_flush();
        lock_request(1,1);
        lock_grant(0);
    }
}
```

One thread per processor (no preemption, no OS)
Loops running in lockstep
One cycle of the loops = one cycle of the Lustre program
Multi-threaded C code

```c
void* thread_cpu0(void* unused){
    lock_init_pe(0); init(); time_init(&time);
    for(;;){
        global_barrier_reinit(2);
        time+=3000; wait(time);
        global_barrier_sync(0);
        dcache_inval();
        f(i,&x);
        dcache_flush();
        lock_grant(1);
        lock_request(0,0);
        dcache_inval();
        h(x,y,&z);
    }
}

void* thread_cpu1(void* unused){
    lock_init_pe(1);
    for(;;){
        global_barrier_sync(1);
        dcache_inval();
        g(z,&y);
        dcache_flush();
        lock_request(1,1);
        lock_grant(0);
    }
}
```

Global barrier synchronization ensures:
- lockstep execution
- real-time period
Multi-threaded C code

void* thread_cpu0(void* unused){
    lock_init_pe(0); init(); time_init(&time);
    for(;;){
        global_barrier_reinit(2);
        time+=3000; wait(time);
        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();
    }
}

void* thread_cpu1(void* unused){
    lock_init_pe(1);
    for(;;){
        global_barrier_sync(1);
        dcache_inval();
        g(z,&y);
        dcache_flush();
        lock_request(1,1);
        lock_grant(0);
    }
}

All remaining code in threads corresponds to data-flow nodes
Multi-threaded C code

void* thread_cpu0(void* unused){
    lock_init_pe(0); init(); time_init(&time);
    for(;;){
        global_barrier_reinit(2);
        time+=3000; wait(time);
        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();
    }
}

void* thread_cpu1(void* unused){
    lock_init_pe(1);
    for(;;){
        global_barrier_sync(1);
        dcache_inval();
        g(z,&y);
        dcache_flush();
        lock_request(1,1);
        lock_grant(0);
    }
}

Hardware lock operations enforce data dependencies inside the cycle
• z not concerned
Multi-threaded C code

```c
void* thread_cpu0(void* unused)
{
    lock_init_pe(0); init(); time_init(&time);
    for(;;)
    {
        global_barrier_reinit(2);
        time+=3000; wait(time);
        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();
    }
}

void* thread_cpu1(void* unused)
{
    lock_init_pe(1);
    for(;;)
    {
        global_barrier_sync(1);
        dcache_inval();
        g(z,&y);
        dcache_flush();
        lock_request(1,1);
        lock_grant(0);
    }
}
```

Explicit cache operations ensure memory coherency
Memory allocation

- Code placement entirely controled
  - 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
Platform API

• Cache coherency
  – dcache_flush – force the write of all dirty lines in the cache/write buffer to memory
  – dcache_inval – invalidate all data cache lines

• Lock synchronization
  – lock_request – request the hardware lock (blocking)
  – lock_grant – grant the hardware lock (non-blocking)

• Time synchronization
  – wait – wait for a specific date

• Global barrier synchronization
  – global_barrier – global barrier of all processors. Exited on all processors at the same time (± a bounded number of CPU cycles)
Timing model

• Analysis of sequential pieces of code
  – In isolation
    • No interferences from concurrent code
  – Need mapping-independent worst-case guarantees
  – Hypotheses on memory allocation, that must be respected during allocation

• Interference model
Analysis of sequential code

- Worst-case execution time (WCET) analysis
  - In our case: aiT from AbsInt
  - Static analysis of sequential functions
    - Assumes no external interferences (timing, synchronization)
    - Can be applied to dataflow nodes
  - For a sequential function f, aiT can compute:
    - WCET(f) = upper bound on the execution time, from function call to return
      - Does not include building the call context
    - WCAT(f,m) = upper bound on the number of memory accesses by f to a memory area m
      - At memory bank input (takes into account cache behavior)
    - WCCAL(f,g) = upper bound on the number of times f calls a library function g
      - Mandatory for us, due to software implementation of division
    - WCSTACK(f) = upper bound on the stack size
Analysis of sequential code

• WCET analysis constraints
  – Analysis is done on statically-allocated code with well-known stack
  – We need allocation-independent values
    • Cache partitioning through strong, architecture-dependent hypotheses on the way mapping is done.
  • Examples on Kalray MPPA256:
    – Allocation of nodes is done with cache line alignment
    – Code and data of all library functions are smaller than 4kbytes
    – Nodes with code or data larger than 4kbytes are aligned on 4kbytes…
    – Specific memory allocation by gcc and custom-made analysis scripts for aiT
Analysis of sequential code

- Remaining thread code is not analyzed using aiT
  - Code snippets
    - Call construction (putting arguments on stack)
    - Cache coherency
    - Synchronization code
    - Global barrier
    - Optional tracing code
  - Instructions not covered or difficult to automate
  - Manual analysis of the code to derive WCET(s), WCAT(s)
    - Hypotheses: No call to library functions, no stack increase
    - Most complex for call construction
Memory interferences

• Request-response protocol
  – Arbitration: Memory requests from multiple sources are arbitrated using a Round Robin policy
  – Atomicity: Once accepted by the arbiter, requests are treated atomically
Memory interferences

• Reads are bursty
  – One-word packet request, 8-word packet response
  – The atomic operation lasts for 8 cycles

• Write operations last for 1 cycle
Memory interferences

- Worst-case interference scenario for two communications
Memory interferences

- Worst-case interference scenario for two communications
  - Tasks $t_1$, $t_2$ accessing concurrently to a memory bank
  - Assume $t_i$ makes $r_i(B)$ read accesses and $w_i(B)$ write accesses to bank $B$, with $a_i(B) = r_i(B) + w_i(B)$
  - An upper bound on the delay $t_2$ imposes on $t_1$ due to interferences on bank $B$ is:
    $$\text{Interf}(t_1, t_2, B) = 8 \times \min(a_1(B), r_2(B)) + \min(a_1(B) - \min(a_1(B), r_2(B)), w_2(B))$$
  - An upper bound for the full interferences on $t_1$ is:
    $$\sum_{\forall B} \sum_{\forall t_j, j \neq i} \text{Interf}(t_1, t_j, B)$$
Architecture description

Architecture

Cores: 2

Memory Excluded
[Start: 0x0000000 End: 0x0600000]
[Start: 0x0c00000 End: 0x1ff0000]
[Start: 0x1ff0000 End: 0x2000000]

Function f:
Text : 104 Data : 0 Stack : 16
WCET : 1174
WCAT :
  Text : [ 2 0 0 ]
  Data : [ 0 0 0 ]
  Stack : [ 0 203 103 ]

- Functional specifications alone are not enough for a real-time implementation
- Specification-dependent input
  - WCET in isolation (pessimistic without context but no interferences)
  - Code size
    - Text
    - Static data
    - Stack usage
  - Number of memory accesses
    - Code, data and stack
    - Triple for code read, data read, and data write
The real-time mapping problem

- Cyclic dependency between mapping and timing analysis
  - How to break this cycle?
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
      – Dependence on mapping may be important (e.g. FFT)
    • What to do in case of non-satisfaction?
  – Use over-approximated timing characterization that cover all possible mappings
    • Produces a safe implementation
      – Our choice
    • Over-approximation costs
      – Need precise timing models for efficient resource allocation
Mapping heuristic

- The base heuristic: **list scheduling**
  - Consider the nodes of the dataflow graph in an order compatible with the intra-cycle data dependencies
  - When considering a node:
    - allocate all data and code it uses onto memory banks
    - allocate it to one of the processing cores
    - choose its start date to ensure that its data dependencies and real-time requirements are met
  - What we need to tune:
    - Choice of a node to schedule between those available at one moment
    - Choice of mapping (allocation and schedule) of the chosen node
    - Ensure that timing accounting remains correct throughout the scheduling process
      - With respect to code generation
    - **Intuitive optimization choices are not the best ones**
Scheduling table

- Reserve time intervals for all function
  - Respect all data dependencies of a cycle
Scheduling table

- Reserve time intervals for all functions
  - Respect all data dependencies of a cycle
  - Reserved(f) = WCET(f) + overheads(f)

- Legend
  - Node call WCET
  - Interferences
  - Memory coherency
  - Synchronization
  - Global barrier
Scheduling table

- Reserved space for a node must account for all overheads
  - Need worst-case bounds on:
    - Synchronization costs
    - Coherency costs
    - Interferences
      - Including by nodes that are not yet scheduled
Synchronization construction

- Objective: Preserve data dependencies and interference pattern
  - Two nodes interfere if they overlap in time and access the same memory bank

- Synchronization synthesis is done after scheduling

- First attempt: minimal synchronization, maximal asynchrony
  - Algorithm based on Lamport clocks
  - Massive use-case parallelism => too many hardware resources needed
Synchronization construction

• Problem of resources
  – Many locks live at the same time
  – Many requests on not granted locks
  – Main reason: nodes with large fan-ins, fan-outs

• Heavy optimizations involving both improved analysis and modifications to scheduling to improve locality of locks
  – Reduction, but not nearly enough. No guarantee of implementability
Synchronization construction

- Solution: sequentialize synchronizations
  - Chains of request-grant before or after node call (plus some optimization)
    - Easy to validate correctness
    - Significantly less synchronization operations
    - Sequencing of operations does not seem penalizing, even for our « fine-grain » parallelism
      - average node WCET = 1000 cycles, hundreds/thousands of nodes
  - Static bound on synchronization overhead:
    - At most two lock requests and two lock grants per node call
Memory coherency

• First attempt: per-data flush and inval operations, with smart ways of optimizing them
  - High cost in code, data, and complexity
• Solution: use the global data cache invalidation and write buffer flush routines
  - Systematic cache invalidation and flush before and after node call respectively
  - (Small) bound on cache coherency costs
  - Architecture-dependent solution!
Interferences

• Need to provision acceptable interferences before scheduling
  – Bound on interferences by not yet scheduled functions
• Increase each WCET by a percentage (e.g. 10%) provisioning interferences
  – Lopht compiler parameter
• When mapping a function during list scheduling, check that its interferences and those of all already mapped functions remain within the predefined bound
  – If not, search for a later date
  – Percentage = 0% => accept no interferences (old Lopht [Carle et al. 2012])
    • Low parallelization
  – Choosing the right value is important for efficiency
Experimental results

- Avionics use-case (Airbus flight controller, DAL A)
  - ~5k unique nodes
  - ~36k variables
- Multi-periodic application
  - Sequential implementation
  - Repeating pattern formed of 5ms « tasks »
  - Each « task » can be represented as a single-period dataflow program
- Our problem:
  - Parallelize each « task »
Experimental results

- One task: 779 nodes, 7943 variables
- Speed-up bound given by critical path: 9.42x
  - Sequential cycle duration/Parallelized cycle duration
  - Infinite number of CPUs, no interferences, no overheads
- Parallelization:
  - 2 CPU: 1.76x
  - 4 CPU: 3.26x
  - 8 CPU: 5.48x
  - 12 CPU: 7.41x
  (cannot use more CPUs due to memory limit, even though we were careful not to waste it)
Experimental results

- Estimating the various overheads:
  - Baseline parallelization on 8 CPU: 5.48x
  - Parallelize while assuming:
    - no interference costs: 6.84x
    - no synchronization overhead: 5.74x
    - no coherency overhead: 5.51x
Experimental results

• Estimating the various overheads:
  – Baseline parallelization on 8 CPU: 5.48x
  – Parallelize while assuming:
    • no interference costs: 6.84x
    • no synchronization overhead: 5.74x
    • no coherency overhead: 5.51x
    • no interference or overhead: 7.99x
      – Embarassingly parallel?
Experimental results

- Embarassingly parallel?
  - Yes – there is a lot of parallelism (9.42x in theory)
  - But exploiting it has a cost in synchronization and (mostly) interferences

- « embarassingly parallel » is not easy to define
  - Depends on application, architecture, mapping
    - E.g. increasing locality using local copies at certain dates reduces interferences
Conclusion

- First real-time implementation method that fully automates timing analysis, in addition to mapping and code generation
  - Real-time systems compilation
  - Relies on strong integration of timing analysis, mapping, code generation, compilation around a precise timing model
  - Works on shared memory multi-cores satisfying certain hypotheses
    - One tile of Kalray MPPA256
  - Good practical results for industrial case studies

- Future work
  - Other platforms
    - Full Kalray MPPA256 chip - code and data overlays and scheduling over NoC
    - Tricore?
  - More native multi-rate support
  - Optimizations
  - Formal validation
Other approaches to code generation

- **Time-triggered**
  - Our first code generation approach for MPPA (dec. 2016)
  - Simpler code
  - Depending on architecture, fine-grain time synchronization may be expensive
    - less overhead on Kalray MPPA256
  - Code is functionally less robust
    - Minor timing errors break the whole execution
      - Functional simulation is impossible with the same code on a different architecture
    - Gains on some functions cannot compensate timing errors on other functions
Other approaches to code generation

● Bulk synchronous parallel (BSP)
  – Separate computations and communications into non-overlapping phases, executed cyclically
  – Timing analysis of computation phases is easy if full spatial isolation is ensured
    • No two processors use the same memory bank => no interferences
    • Full spatial isolation => memory&communication costs
  – WCET analysis of communication phases remains complicated
  – Scheduling dataflow specifications for BSP is non-trivial
    • Trade-off between parallelization and latency in the construction of computation phases
Heuristics vs « exact » methods

- Constraint solving, SMT, ILP
  - Popular in real-time scheduling
- Our problem can be put in this form
  - Previous attempts on simpler problems [FORMATS'15] – not scalable
  - Recent advances in solver technology
  - Problem far more complex: allocation of code data, interferences, scheduling, etc.
- Difficult to predict how much time it will take (or if it terminates)
  - What to do when it does not?
Reused results

• Previous work
  - [Carle et al. 2012] – Mapping into shared-memory many-cores without memory interferences
  - [Puaut&Potop 2013] – WCET analysis of synchronous parallel code without memory interferences
  - [Rihani et al. 2016] – Timing analysis on Kalray MPPA256 in the presence of memory interferences
Cannot use OS-like semaphores due to HW abstraction with high cost (e.g. critical sections, etc.)
Hypotheses on platform and external code (nodes+libs)

• Platform API
  – dcache_flush, dcache_inval
  – lock_request, lock_grant
  – wait
  – global_barrier

• Node call conventions

• Memory allocation conventions for nodes and libs