A compilation-like approach to real-time systems implementation

Keryan Didier
11/30/17
The need for automation

- Higher level program specification
- Implementation automation
- Formalism and certification

More effort at high level,
More functionality

Engineer struggling to keep her code from crashing, 1969
Standard compilation

Functionality (C program)

Compiler Linker

Sequential executable code
Standard compilation

Functionality (C program)

Compiler Linker

Sequential executable code

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

- Functionality (Lustre program)

- Platform model, Allocation

- (Parallelization) Lustre compiler
  C compiler
  Linker

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

Non-functional requirements

Functionality (Lustre program)

Parallel platform model

Parallelisation
Real-time scheduling
Lustre compiler
C compiler
Linker

Parallel or sequential real-time executable code
Outline

• Data-flow programming in Lustre
  – And timing extensions
• Structure of an implementation
• Timing model
• Resource allocation and code generation
  – Also known as compilation
• Experimental results
• Conclusion
Data-flow programming

```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
```

- Well formed concurrent code
  - Cyclic execution
  - Static Single Assignment
  - No recursion, no heap
Data-flow programming

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

- Well formed concurrent code
  - Cyclic execution
  - Static Single Assignment
  - No recursion, no heap

- Natural concurrent semantics
- Well understood compilation and analysis
Data-flow programming

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
Non-functional requirements

- Real-time requirements
  - Cycle period
  - Release dates
  - Deadlines

- Other requirements
  - Allocation constraints

```javascript
period(3000)
node main () returns ()
var
  i : int; x : float;
  y : int; z : int;
  d : int;
let
  i = read_int();
  deadline(1500) x = f(i);
  y = g(d);
  z = h(x, y);
  d = 0 fby z;
  () = write_int(z);
```
Structure of an implementation

- Multi-threaded C code
  - Initialization
  - Node 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

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);
    }
}
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);
    }
}

One cycle of the loop = one cycle on the synchronous model
- Global barrier synchronization
- Real-time period
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

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

Hardware lock operations enforce data-flow 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

```c
.x = 0x88e88;
```
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
Timing model

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

• Interference model
Timing model

- **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
Timing model

• 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
Timing model

- 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

- Data 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 $t_1$, $t_2$ acceding 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: 0x000000 End: 0x060000]
[Start: 0x0c0000 End: 0x1ff000]
[Start: 0x1ff000 End: 0x200000]

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 characteristics
  - 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
      – Large systems featuring significant interferences (e.g. FFT)
    • What to do in case of non-satisfaction?
  – Use over-approximated timing characterization that cover all possible mappings
    • Better support for full automation
      – 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)
    • Ensure that timing accounting remains correct throughout the scheduling process
    • Intuitive optimization choices are not the best ones
Scheduling table

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

- Reserve time intervals for all nodes
  - Respect all data dependencies of a cycle
  - \(\text{Reserved}(f) = \text{WCET}(f) + \text{overheads}(f)\)

- 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

- Actual synthesis of synchronization code is done after the scheduling phase is completed
  - Based on absolute timing of the scheduling
- Earlier work: minimal synchronization, maximal asynchrony
  - Implement only the synchronizations imposed by functional correctness and preservation of interferences
  - Massive application 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

- Our 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 1000 cycles per node call, hundreds/thousands of nodes

- Static bound on synchronization overhead:
  - At most two lock requests and two lock grants per node call
Memory coherency

• Original choice: per-data flush and inval operations, with smart ways of optimizing them
  − High cost in code, data, and complexity
• Our solution on Kalray MPPA256: 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
Interferences

- Need to provision acceptable future interferences before execution
- Increase each WCET by a percentage (e.g. 10%) provisioning interferences
  - Lopht compiler parameter
- When mapping a node during list scheduling, check that its interferences and those of all already mapped nodes remain within the predefined bound
  - If not, search for a later date
    - Create synchronization for timing rather than functionality
  - Percentage = 0% => accept no interferences (old Lopht [Carle at 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 edges
- Multi-periodic application
  - Sequential implementation
  - Repeating pattern formed of 5ms « tasks »
  - Nodes of a « task » can be represented as a single-period dataflow program
- Our problem:
  - Parallelize each « task »
Experimental results

• One task: 779 nodes, 7943 edges
• Theoretical parallelization bound given by critical path (infinite number of CPUs, no interferences, no overheads): 9.42x

• 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

• The various costs:
  – 8 CPU: 5.48x
  – assume no interference costs: 6.84x
    • Interference model from Verimag
  – assume no sync overhead: 5.74x
    • 2*wait+2*signal = 70 cycles per node
  – assume no cache overhead: 5.51x
    • 1*inval+1*flush = 36 cycles per node
  – assume no interference or overhead: 7.99x
Conclusion

• Lopht currently allows the mapping of single-rate Lustre onto one tile of Kalray MPPA256
  – Multi-rate by hyper-period expansion

• Future works
  – Full Kalray MPPA256 chip
    • Code and data overlays and scheduling over NoC
      – Reuse previous results
    • More native handling of multi-period
  – Formal validation
  – Optimizations
Previous and related work

• Previous work
  – [Rihani et al. 2016] – Timing analysis on Kalray MPPA256 in the presence of memory interferences

• Related work
  – Lots of work on parallel and real-time application mapping based on static scheduling
  – A few paper on timing analysis for parallel code
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
Cannot use OS-like semaphores due to HW abstraction with high cost (e.g. critical sections, etc.)