The Kiel Lustre Processor/ WCRT Interface Algebra

Claus Traulsen

Joint work with Michael Mendler (U. Bamberg), Reinhard von Hanxleden

Christian-Albrechts Universität zu Kiel

Synchron 2008

1. December 2008

Institut
für Informatik
Christian-Albrechts-Universität zu Kiel

Outline

- Kiel Lustre Processor
  - Architecture
  - Compilation
- Interface Types for WCRT Analysis
  - WCRT Analysis
  - Interfaces

Reactive Processing

- Key Observation: Reactive control flow is not matched by common processors.
- Reactive processors: ISA tailored to reactive control flow

- Inspiration for ISA: synchronous languages

KEP	Esterel	Multi-threaded
KReP	Lustre	Multi-core
KLP	Lustre	Multi-threaded

Advantages

- Deterministic behavior
- Precise Timing
  Simplify WCRT analysis
- Resource usage
    - Power consumption
    - Resource per high-level operation
- Dependability
- Smaller programs
- Valid programs

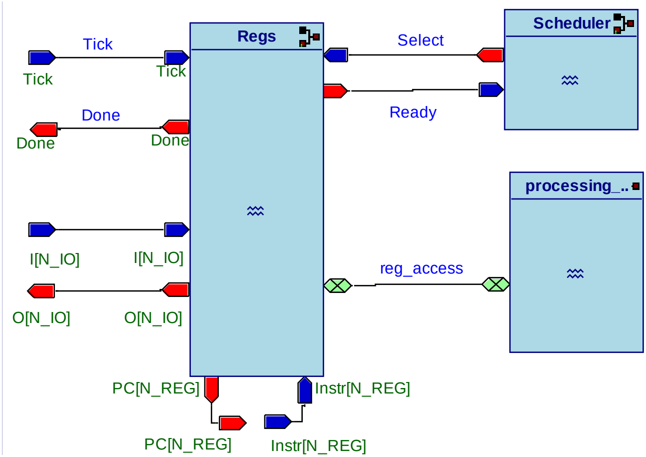
Kiel Lustre Processor

- Dedicated processor for Lustre programs
- Make use of dataflow nature → explicit parallelism
- Compare Multiple-execution units vs. multi-core
- Directly support clocks: What needs to be executed?
- Support automata (SCADE)

Related Work

- Lustre execution and compilation (Raymond, *et al.*)
- Distribution of synchronous programs (Girault, *et al.*)
- Dataflow processing (*e. g.*, Manchester machine, Gurd, *et al.*)
- Precision time architecture (PReT) (Edwards, Lee, *et al.*)
- StarPro (Roop, *et al.*)

Architecture Overview



Registers

Each register holds a Lustre equation:

- current value
- previous value
- program-counter
- clock id

Additional information:

- next instruction
- done flag

```
...
C=0->not pre(C);
N=X+1 when C;
...
```

↓

5:	1	0	0	L_C
6:	9	3	5	L_N

Execution

```
each TICK do
  ?pre_value <= ?value;
  ?done <= false;
  while(not done_all)
    select i with ready[i]
      execute i
  ||
    ?done[j] <= true if ?done[clock[j]]
                      and value[clock[j]]=0
  ||
    ?ready[j] <= done[args(j)]
  end
end loop
```

Ready:

- prefetch instruction
- sense clock
- sense arguments

Execution Example

```
1: C = false -> not pre(C);
2: NC = not C;
3: A = (I+1) when C;
4: B = (I-1) when NC;
```

Reg	val	pval	PC	done	ready
I	12	16	$\perp$	1	0
C	1	1	L_C	0	1
NC	0	0	L_NC	0	0
A	17	17	L_A	0	0
B	5	5	L_B	0	0

Scheduling

- Lustre programs are acyclic ⇒ static schedule possible
- Benefits of dynamic schedule:
    - Data dependent parallelism
    - Clocks can be tested in parallel

Instruction Set Architecture

- Standard arithmetical operations
- `INIT` initialize registers: clock and PC
- `INPUT`, `OUTPUT` connect register to IO
- `DONE` mark current computation as finished and set start-point for next tick

Counter example:

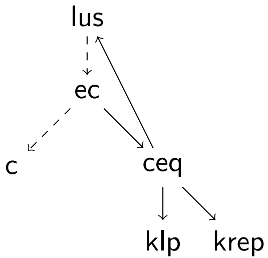
```
   INPUT      R
   INPUT      X
   OUTPUT     C
   INIT       C     L_C
   DONE

 L_C:
   IVMOV C    0
   DONE L_C_run
 L_C_run:
   IXOR _C_0  pre(X) 1
   JF R      L_L_0
   IVMOV C    0
   JMP  L_L_1
 L_L_0:
   AND  temp  X     _C_0
   ...
```

Lustre Compilation (as far as I understood it . . .)

- Expand nodes, arrays, . . .
- Check types and clocks
- Order equations (find schedule)
- Extract common expression
- Compute necessary registers
- Implement equations, replace `when` by `if`

KLP Compilation



lus	Lustre Source file
ec	expanded code
c	generated C-Code
ceq	clocked equation
klp	single core, dynamic schedule
krep	multi-core, static schedule

Clocked Equations

- Equation `x=current(init -> e) when C`

  `init` initial value
  `e` simple expression
  `C` Clock

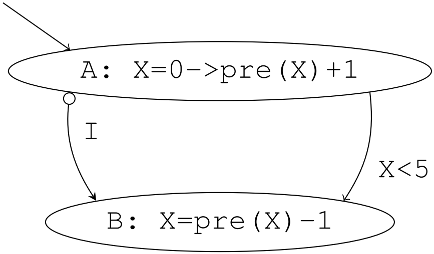
- No nesting of `pre`
- No additional clock operators
- Simple clocks
- Still valid Lustre code

→ direct map to KLP ISA

Compilation of Automata

1. Compilation via Lustre
   - + no additional effort
   - – suboptimal efficiency
2. Direct compilation
   - + only effects compiler
   - – transitions are always tested
3. Using watcher
   - + test all transitions in parallel
   - – additional hardware needed (okay for weak abort → extend `DONE` instruction)
   - – abortion of parallel parts

Direct Compilation of Automata



```
   INIT X L_X

X: IMOV X 0
   DONE L_A
A: // Check strong abort
  JT I L_B
  // Compute
  IADD X X 1
  // Check weak abortion
  GEI T X 5
  JT T LT
  DONE LA
T: DONE LB
  ...
```

Watcher

- New instruction `WATCH Reg, PC1, PC2`
  - Address range
  - Register to watch
  - Replacement address
- When executing code in address-range and register is true: execute replacement
- Replacement code reinitialize registers
- Additional dependencies for the scheduler
- Not yet implemented

What's the Gain?

Deterministic behavior	Yes
Precise Timing	WCRT analysis still missing
Resource usage	Need more tests
Dependability	Not at all
Smaller programs	Yes
Valid programs	Yes, but not supported by compiler

Conclusion (1. Part)

- Reactive processing from Lustre
- Direct use of parallelism
- Natural synchronization points: tick
- Number of used values for one tick is fixed

Outline

Kiel Lustre Processor

- Architecture
- Compilation

Interface Types for WCRT Analysis

- WCRT Analysis
- Interfaces

WCRT vs. WCET

Worst Case Execution Time

- Compute maximal execution time for piece of code

Worst Case Reaction Time

- Compute maximal time to react:
  one valid program state to another
- Similar to stabilization time of circuits

The KEP and its WCRT

```
loop
 abort
  [await A || await B];
  emit O;
  halt
 when R
end loop
```

- 1 cycle/instruction
- WCRT: count instructions

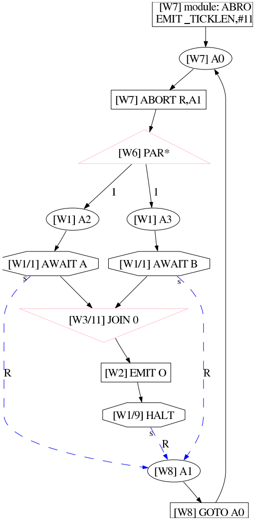
```
     EMIT_TICKLEN,#11
A0:  ABORT R,A1
     PAR 1,A2,1
     PAR 1,A3,2
     PARE A4,1

A2:  AWAIT A
A3:  AWAIT B
A4:  JOIN 0
     EMIT O
     HALT
A1:  GOTO A0
```

WCRT as Longest Path

```
    EMIT _TICKLEN,#11
A0: ABORT R,A1
    PAR 1,A2,1
    PAR 1,A3,2
    PARE A4,1
A2: AWAIT A
A3: AWAIT B
A4: JOIN 0
    EMIT O
    HALT
A1: GOTO A0
```

- Implemented by Marian Boldt [SLA++P'07]
- Compute longest path between delay-nodes
- Abstract data-dependencies
- *Ad-hoc* optimizations



Interfaces

- Use interface algebra to express WCRT
- Solid theoretical basis
- Modular computation (dynamic programming)
- Computation: $(max, +)$-agebra on timing matrix
- Refinement (Data-dependencies)

Interface Types

$$D{:}\phi \supset \psi$$

- Delay Matrix

$$D = \begin{pmatrix} d_{11} & \cdots & d_{1n} \\ \vdots & & \vdots \\ d_{m1} & \cdots & d_{mn} \end{pmatrix}$$

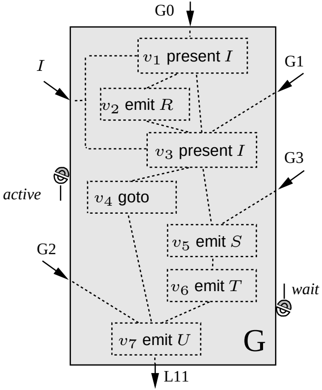
- Input Control

$$\phi = \zeta_1 \vee \zeta_2 \vee \cdots \vee \zeta_m$$

- Output Control

$$\psi = \circ\zeta_1 \oplus \circ\zeta_2 \oplus \cdots \oplus \circ\zeta_n$$

Expressing the WCRT



- $(6) : G0 \supset \circ L11$
- $(6, 4, 3, 1)$: $(G0 \vee G1 \vee G3 \vee G2) \supset \circ L11$
- $(5, 5, 3, 4, 3, 1)$: $((G0 \wedge I) \vee (G0 \wedge \neg I) \vee (G1 \wedge I) \vee (G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11$
- $(5, 3, 4, 3, 1) : (G0 \vee (G1 \wedge I) \vee (G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11$
- $(5, 3, 4, 1) : (G0 \vee ((G1 \wedge I) \oplus G3) \vee (G1 \wedge \neg I) \vee G2) \supset \circ L11$
- $(5) : G0 \supset \circ L11$

Conclusion (2. Part)

- Flexible: Set degree of exactness
- Benefits:
  - Handling of control data
  - Systematic treatment of parallel execution
- Implemented some of the basic ideas:
  Promising first results
- See DATE'09

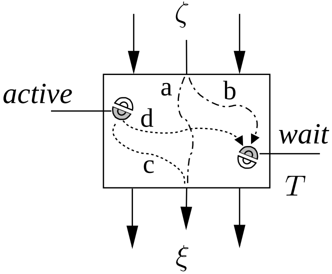
Outlook

KLP
- Multicore version
- Non-Boolean clocks
- Handle automata
- Direct translation from SCADE

WCRT
- Implementation
- Delayed abortion + traps
- Consider Thread priorities
- Formal semantics of the KEP

Thanks for your attention!

Types for Nodes



$$T \quad = \quad \begin{pmatrix} d_{thr} & d_{src} \\ d_{snk} & d_{int} \end{pmatrix} : (\zeta \vee active) \supset (\circ\xi \oplus \circ wait)$$