HW/SW Co-Design for Esterel Processing

Sascha Gädtke  Claus Traulsen  Reinhard von Hanxleden

Real-Time Systems and Embedded Systems
Department of Computer Science
Christian-Albrechts-University Kiel

Codes+ISSS 2007
Outline

Introduction
- The Kiel Esterel Processor
- Signal Expressions

HW/SW Co-Design
- Partitioning
- Hardware/Software Synthesis
- Interface

Results and Further Work
Esterel

- Non-standard control flow in hard real-time systems:
  - Concurrency
  - Preemption
  - Time critical

- Esterel: a synchronous reactive language
  - Deterministic behavior
  - Clean mathematical semantics
  - Discrete timing model

- Implementation:
  - Compilation to Software (e.g., C)
  - Synthesize Hardware (VHDL)
  - Execute on a reactive processor
Esterel

- Non-standard control flow in hard real-time systems:
  - Concurrency
  - Preemption
  - Time critical
- Esterel: a synchronous reactive language
  - Deterministic behavior
  - Clean mathematical semantics
  - Discrete timing model
- Implementation:
  - Compilation to Software (e.g., C)
  - Synthesize Hardware (VHDL)
  - Execute on a reactive processor
Esterel

- Non-standard control flow in hard real-time systems:
  - Concurrency
  - Preemption
  - Time critical
- Esterel: a synchronous reactive language
  - Deterministic behavior
  - Clean mathematical semantics
  - Discrete timing model
- Implementation:
  - Compilation to Software (e.g., C)
  - Synthesize Hardware (VHDL)
  - Execute on a reactive processor
The Abro Example

module ABRO:
  input A, B, R;
  output O;

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

end module

▶ Wait simultaneously for A and B
▶ When both have occurred, emit O
▶ Reset behavior by R
The Abro Example

module ABRO:
  input A, B, R;
  output O;

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

end module

- Wait simultaneously for A and B
- When both have occurred, emit O
- Reset behavior by R
The Abro Example

```plaintext
module ABRO:
    input A, B, R;
    output O;

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

- Wait simultaneously for A and B
- When both have occurred, emit O
- Reset behavior by R
The Abro Example

```
module ABRO:
  input A, B, R;
  output O;

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

end module
```

- Wait simultaneously for A and B
- When both have occurred, emit O
- Reset behavior by R
The Abro Example

module ABRO:
  input A, B, R;
  output O;

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

- Wait simultaneously for A and B
- When both have occurred, emit O
- Reset behavior by R
The Abro Example

```plaintext
module ABRO:
    input A, B, R;
    output O;

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

end module
```

- Wait simultaneously for A and B
- When both have occurred, emit O
- Reset behavior by R
The Abro Example

module ABRO:
input A, B, R;
output O;

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

- Wait simultaneously for A and B
- When both have occurred, emit O
- Reset behavior by R
The Abro Example

```plaintext
module ABRO:
    input A, B, R;
    output O;

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

end module
```

- Wait simultaneously for A and B
- When both have occurred, emit O
- Reset behavior by R

---

![Timing diagram showing the Abro example with inputs B, A, R, and outputs O.](image)
The Abro Example

module ABRO:
    input A, B, R;
    output O;

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

- Wait simultaneously for A and B
- When both have occurred, emit O
- Reset behavior by R
Signals

module example:
input A, B;
output O1;
input {D,E,F} := false : bool;
output O2: bool;

signal C in
  present (A or B) and C then
    emit O1
  end present;
  if (?D or ?E) and ?F then
    emit O2(?D)
  end if
end signal
end module

- Communication
  - with the environment
  - internally

- Pure signals: present or absent

- Valued signals: carry an additional value

- Status and value consistent within one tick

- Can test for expression on signals

- ... or expression on signal values
module example:
  input A, B;
  output O1;
  input {D,E,F} := false : bool;
  output O2: bool;

  signal C in
    present (A or B) and C then
      emit O1
    end present;
    if (?D or ?E) and ?F then
      emit O2(?D)
    end if
  end signal
end module

- **Communication**
  - with the environment
  - internally
- **Pure signals**: present or absent
- **Valued signals**: carry an additional value
- Status and value consistent within one tick
- Can test for expression on signals
- ... or expression on signal values
The Kiel Esterel Processor

```plaintext
loop
  abort
  [await A || await B];
  emit 0;
  halt
  when R
end loop

⇓

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 0
  HALT
A1: GOTO A0
```

- ISA inspired by Esterel
- Direct execution of most statements
- Implementation of concurrency by multiple threads
- Avoid jitter by stalling for known time
- More flexible than hardware
- Faster than software on general purpose processor
The Kiel Esterel Processor

loop
  abort
    [await A || await B];
  emit 0;
  halt
when R
end loop

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 0
     HALT
A1:   GOTO A0

- ISA inspired by Esterel
- Direct execution of most statements
- Implementation of concurrency by multiple threads
- Avoid jitter by stalling for known time
- More flexible than hardware
- Faster than software on general purpose processor
Compiling Signal Expressions

**Problem:** Complex Signal Expressions are sequentialized into multiple KEP Assembler instructions

```plaintext
present (A or B) and C then
```

```plaintext
⇒
PRESENT C, A0
PRESENT A, A2
GOTO A1
A2: PRESENT B, A0
```

**Solution:** Calculate Expressions in logic block connected to the KEP

- Sequential logic much faster than execution of one instruction
- Reduce total number of instructions → reduce Ticklength
- Faster execution of every tick
Compiling Signal Expressions

Problem: Complex Signal Expressions are sequentialized into multiple KEP Assembler instructions

```
present (A or B) and C then
```

Solution: Calculate Expressions in logic block connected to the KEP

- Sequential logic much faster than execution of one instruction
- Reduce total number of instructions → reduce Ticklength
- Faster execution of every tick
Outline

Introduction
  The Kiel Esterel Processor
  Signal Expressions

HW/SW Co-Design
  Partitioning
  Hardware/Software Synthesis
  Interface

Results and Further Work
Overview

1. Identify expressions
2. Replace expressions by new signals
3. Define new modules to compute these
4. Synthesize modules to hardware
5. Compile remaining program to KEP assembler
6. Connect hardware to KEP
7. Power on
Overview

1. Identify expressions
2. Replace expressions by new signals
3. Define new modules to compute these
4. Synthesize modules to hardware
5. Compile remaining program to KEP assembler
6. Connect hardware to KEP
7. Power on
Identify Expressions

module example:
  input A, B;
  output 01;
  input {D,E,F} := false : bool;
  output 02: bool;

signal C in
  present (A or B) and C then
    emit 01
  end present;
  if (?D or ?E) and ?F then
    emit 02(?D)
  end if
end signal
end module

- Extract signal expressions
- Extract Boolean parts from expressions

if (?D or ?E) and ?V>5 then
Identify Expressions

module example:
input A, B;
output O1;
input {D,E,F} := false : bool;
output O2: bool;
signal C in
    present (A or B) and C then
        emit 01
    end present;
    if (?D or ?E) and ?F then
        emit O2(?D)
    end if
end signal
end module

- Extract signal expressions
- Extract Boolean parts from expressions
Identify Expressions

module example:
input A, B;
output O1;
input \{D,E,F\} := \text{false} : \text{bool};
output O2: \text{bool};

signal C in
  present (A or B) and C then
    emit O1
  end present;
if (?D or ?E) and ?F then
  emit O2(?D)
end if
end signal
end module

- Extract signal expressions
- Extract Boolean parts from expressions

if (?D or ?E) and ?\text{V}>5 then
Replace expressions

signal C in
    present (A or B) and C then
        emit O1
    end present;
    if (?D or ?E) and ?F then
        emit O2(?D)
    end if
end signal
end module

⇒

... signal C, A_OR_B_AND_C in
    trap COSYN_TRAP_0 in
        run intro_example_hw_2
    ||
        present A_OR_B_AND_C then
            emit O1
        end present;
        if (?D_OR_E_AND_F) then
            emit O2(true)
        end if;
        exit CONSYN_TRAP_0
    end trap;
end signal;
...
Compute Signals in New Modules

```
signal C in
  present (A or B) and C then
    emit 01
  end present;
  if (?D or ?E) and ?F then
    emit 02(?D)
  end if
end signal

⇒

module example_hw_2:
  input A, B, C;
  output A_OR_B_AND_C
  every immediate [(A or B) and C] do
    emit A_OR_B_AND_C
  end every
end module

⇒

module example_hw_1:
  input {D,E,F} := false : bool;
  output D_OR_E_AND_F : bool;
  sustain D_OR_E_AND_F(?D or ?E and ?F)
end module
```
Compute Signals in New Modules

```vhdl
module example_hw_2:
  input A, B, C;
  output A_OR_B_AND_C
  every immediate [(A or B) and C] do
    emit A_OR_B_AND_C
  end every
end module

module example_hw_1:
  input {D,E,F} := false : bool;
  output D_OR_E_AND_F : bool;
  sustain D_OR_E_AND_F(?D or ?E and ?F)
end module
```
Synthesize to Hardware

module example_hw_1:
input {D,E,F} := false : bool;
output D_OR_E_AND_F : bool;
sustain D_OR_E_AND_F(?D or ?E and ?F)
end module

module example_hw_2:
input A, B, C;
output A_OR_B_AND_C
every immediate [A or B and C] do
  emit A_OR_B_AND_C
end every
end module

entity intro_example is
port(A: in std_logic;
    D: in std_logic_vector(1 downto 0); ...
);
end intro_example
architecture intro_example_BEH of intro_example is
begin
  D_OR_E_AND_F(1) <= (D(1) or (E(1) and F(1)));
  A_OR_B_AND_C <= (A or B) and C;
end intro_example_BEH;
Compile to KEP Assembler

```plaintext
signal C, A_OR_B_AND_C in
trap COSYN_TRAP_0 in
  run intro_example_hw_2
  ||
  present A_OR_B_AND_C then
    emit 01
  end present;
  if (?D_OR_E_AND_F) then
    emit 02(true)
  end if;
  exit CONSYN_TRAP_0
end trap;
end signal;
...
```

```plaintext
SIGNAL C
SIGNAL A_OR_B_AND_C
PRESENT A_OR_B_AND_C, A0
EMIT 01
A0: LOAD REG0, ?D_OR_E_AND_F
   CMPS REG0, #1
   JW EE, A1
   EMIT 02, #1

► New signal is local
► No run
► No trap
```
Interface to the Logic

Interface block stores I/O and local signals

- Connect logic to SinotReg
- Direct access to signal status → change status within a tick
- External logic needn’t care about suspension
Valued Expressions

- No arithmetic or comparison → too expensive
- SDatBoolean replicates last bit of value
- Access like pure signal
Correctness

**Complete Chain:**
- Tested on different benchmarks
- Checked executed behavior

**Partitioning:**
- Verified partitioning by sequential equivalence check
- Partitioning looks okay

**Interface:**
- Computation faster than instruction clock
- No read/write conflict
- No reuse of interface signals
Correctness

Complete Chain:
- Tested on different benchmarks
- Checked executed behavior

Partitioning:
- Verified partitioning by sequential equivalence check
- Partitioning looks okay

Interface:
- Computation faster than instruction clock
- No read/write conflict
- No reuse of interface signals
Correctness

Complete Chain:  
- Tested on different benchmarks
- Checked executed behavior

Partitioning:  
- Verified partitioning by sequential equivalence check
- Partitioning looks okay

Interface:  
- Computation faster than instruction clock
- No read/write conflict
- No reuse of interface signals
Outline

Introduction
  The Kiel Esterel Processor
  Signal Expressions

HW/SW Co-Design
  Partitioning
  Hardware/Software Synthesis
  Interface

Results and Further Work
Experimental Results

- No signal expression $\rightarrow$ no benefit
- Choose several Benchmarks, with reasonable number of signal expressions
- Compute WCRT and measure actual execution times
- Need new local signals
- Number of additional slices small
- Power consumption per Tick proportional to Ticklength
Experimental Results (cont)

Token Ring 10

- WCRT: 256, 208, 48
- ACET: 172, 148, 24
- MAX: 201, 155, 46
- Local Sig.: 51, 61, -10

Backhoe

- WCRT: 209, 86, 123
- ACET: 16, 9, 7
- MAX: 32, 22, 10
- Local Sig.: 3, 5, -2
Conclusion

- Simple approach for HW/SW Co-Design
  1. Partition on Esterel level
  2. Implement some modules in HW, some in SW
- Esterel’s clear semantics makes partitioning easy
- Concurrency and preemption control is executed efficiently on the KEP
  → HW modules can be simple
- Actual benefit depends highly on program
- Fully Implemented (extension to CEC)
Further Work

- Correctness
- KEP running on a faster clock
  - Improved scheduling
  - How to guarantee correctness?
- Beyond expressions
- Co-Processor
- ...