Multi-Threaded Reactive Processing

Xin Li  Marian Boldt  Reinhard v. Hanxleden

Real-Time Systems and Embedded Systems Group
Department of Computer Science
Christian-Albrechts-Universität zu Kiel, Germany
www.informatik.uni-kiel.de/rtsys

SYNCHRON’06
November 2006
Why “Reactive Processing”? 

Control flow on traditional (non-embedded) computing systems: 
- Jumps, conditional branches, loops 
- Procedure/method calls 

Control flow on embedded, reactive systems: all of the above, plus 
- Concurrency 
- Preemption 

The problem: mismatch between traditional processing architectures and reactive control flow patterns 
- Processing overhead, e.g. due to OS involvement or need to save thread states at application level 
- Timing unpredictability
Why “Reactive Processing”?

Control flow on traditional (non-embedded) computing systems:
- Jumps, conditional branches, loops
- Procedure/method calls

Control flow on embedded, reactive systems: all of the above, plus
- Concurrency
- Preemption
Why “Reactive Processing”? 

Control flow on traditional (non-embedded) computing systems: 
- Jumps, conditional branches, loops 
- Procedure/method calls 

Control flow on embedded, reactive systems: all of the above, plus 
- Concurrency 
- Preemption 

The problem: mismatch between traditional processing architectures and reactive control flow patterns 
- Processing overhead, e.g. due to OS involvement or need to save thread states at application level 
- Timing unpredictability
Why bother?

Reactive processing yields

- Low power requirements
- Deterministic control flow
- Predictable timing
- Short design cycle
Why bother?

Reactive processing yields

▶ Low power requirements
▶ Deterministic control flow
▶ Predictable timing
▶ Short design cycle

Can use reactive processor

▶ in stand alone, small reactive applications
Why bother?

Reactive processing yields

- Low power requirements
- Deterministic control flow
- Predictable timing
- Short design cycle

Can use reactive processor

- in stand alone, small reactive applications
- as building block in SoC designs

Why bother?

Reactive processing yields

- Low power requirements
- Deterministic control flow
- Predictable timing
- Short design cycle

Can use reactive processor

- in stand alone, small reactive applications
- as building block in SoC designs
Reactive Processing Part I: The Language

Have chosen **Esterel**:  
- Created in the early 1980's  
- For programming control-dominated reactive systems  
- Used as intermediate language for Statecharts (Safe State Machines)  
- Textual imperative language with reactive control flow constructs  
  - Concurrency  
  - Weak/strong abortion  
  - Exceptions  
  - Suspension  
- A synchronous language  
- Deterministic behavior, clean semantics  
- Currently undergoing IEEE standardization (v7)
Reactive Processing Part II: The Execution Platform
Reactive Processing Part II: The Execution Platform

Hardware
- Custom Hardware
- Environment

Software
- COTS Assembler
- COTS-μC
- Environment

Co-design
- COTS Assembler
- COTS-μC
- Custom Hardware
- Environment

Patched Processor
- Extended Assembler
- PIC Core
- Extension
- Environment

Esterel Processor
- Esterel Assembler
- Esterel-μC
- Environment
Related Work/Contributions

RePIC [Roop et al.’04]/EMPEROR [Yoong et al.’06]
  ▶ Multi-processing patched reactive processor
  ▶ Three-valued signal logic + cyclic executive
Related Work/Contributions

RePIC [Roop et al.’04]/EMPEROR [Yoong et al.’06]
- Multi-processing patched reactive processor
- Three-valued signal logic + cyclic executive

Kiel Esterel Processor 1–3
- Multi-threading custom reactive processor
- Provides most Esterel primitives, but still incomplete
- No compilation scheme to support concurrency
Related Work/Contributions

RePIC [Roop et al.’04]/EMPEROR [Yoong et al.’06]
- Multi-processing patched reactive processor
- Three-valued signal logic + cyclic executive

Kiel Esterel Processor 1–3
- Multi-threading custom reactive processor
- Provides most Esterel primitives, but still incomplete
- No compilation scheme to support concurrency

KEP3a (this work)
- Provides all Esterel primitives
- Refined ISA
- Compiler exploits multi-threading
Overview

Introduction

The Kiel Esterel Processor
  The Esterel Language
  Instruction Set Architecture
  Processor Architecture

The Compiler

Wrap-Up
The Esterel Language

Logical Ticks

- Execution is divided into ticks
- Synchrony hypothesis:
  Outputs generated from given inputs occur at the same tick

Signals

- Present or absent throughout a tick
- Used to communicate internally and with the environment
The Esterel Language

Logical Ticks

- Execution is divided into \textit{ticks}
- \textbf{Synchrony hypothesis:}
  
  Outputs generated from given inputs occur at the same tick

Signals

- \textit{Present or absent} throughout a tick
- Used to communicate internally and with the environment

```
module ABRO:
    input A, B, R;
    output O;
    loop
        abort
        [ await A
         ||
         await B ];
        emit O
        halt;
    when R
    end loop;
end module
```
The Esterel Language

Logical Ticks
- Execution is divided into *ticks*
- **Synchrony hypothesis:** Outputs generated from given inputs occur at the same tick

Signals
- *Present* or *absent* throughout a tick
- Used to communicate internally and with the environment

```esterel
module ABRO:
  input A, B, R;
  output O;
  loop
    abort
    [ await A ||
      await B ];
    emit O
    halt;
  when R
  end loop;
end module
```
The Esterel Language

Logical Ticks

- Execution is divided into *ticks*

- **Synchrony hypothesis:** Outputs generated from given inputs occur at the same tick

Signals

- *Present* or *absent* throughout a tick

- Used to communicate internally and with the environment

```plaintext
module ABRO:
  input A, B, R;
  output O;
  loop
    abort
    [ await A ||
      await B ];
    emit O
    halt;
    when R
  end loop;
end module
```
The Esterel Language

Logical Ticks

- Execution is divided into *ticks*
- **Synchrony hypothesis:**
  Outputs generated from given inputs occur at the same tick

Signals

- *Present* or *absent* throughout a tick
- Used to communicate internally and with the environment

```plaintext
module ABRO:
  input A, B, R;
  output O;
  loop
    abort
    [ await A ||
      await B ];
    emit O
    halt;
  when R
  end loop;
end module
```
The Esterel Language

Logical Ticks

- Execution is divided into *ticks*

- **Synchrony hypothesis:** Outputs generated from given inputs occur at the same tick

Signals

- *Present* or *absent* throughout a tick

- Used to communicate internally and with the environment

```plaintext
module ABRO:
    input A, B, R;
    output O;
    loop
        abort
        [ await A
        ||
        await B ];
        emit O
        halt;
    when R
    end loop;
end module
```
The Esterel Language

Logical Ticks

- Execution is divided into *ticks*
- **Synchrony hypothesis:** Outputs generated from given inputs occur at the same tick

Signals

- *Present* or *absent* throughout a tick
- Used to communicate internally and with the environment

```plaintext
module ABRO:
  input A, B, R;
  output O;
  loop
    abort
    [ await A ||
      await B ];
    emit O
    halt;
  when R
  end loop;
end module
```
Candidates for the Instruction Set

Esterel kernel statements

- ||
- suspend ... when S
- trap T in ... exit T ... end trap
- pause
- signal S in ... end
- emit S
- present S then ... end
- nothing
- loop ... end loop
- ;
Candidates for the Instruction Set

Esterel kernel statements

- ||
- suspend ... when S
- trap T in ... exit T ... end trap
- pause
- signal S in ... end
- emit S
- present S then ... end
- nothing
- loop ... end loop
- ;

Derived statements

- [weak] abort ... when S
- await S
- ...
The KEP Instruction Set

- Includes all kernel statements
- In addition, some derived statements

This redundancy improves space/time efficiency
The KEP Instruction Set

- Includes all kernel statements
- In addition, some derived statements

*This redundancy improves space/time efficiency*

```
T0S: % trap T in
A0: % loop
    PAUSE % pause;
    PRESENT S,A1 % present S then
    EXIT T0E, T0S % exit T
A1: % end present
    GOTO A0 % end loop
T0E: % end trap;
```
The KEP Instruction Set

- Includes all kernel statements
- In addition, some derived statements

*This redundancy improves space/time efficiency*

```
T0S: % trap T in
A0: % loop
  PAUSE % pause;
  PRESENT S,A1 % present S then
  EXIT T0E, T0S % exit T
A1: % end present
  GOTO A0 % end loop
T0E: % end trap;
```
The KEP Instruction Set

- Includes all kernel statements
- In addition, some derived statements

*This redundancy improves space/time efficiency*

```
T0S: % trap T in
A0: % loop
    PAUSE % pause;
    PRESENT S,A1 % present S then
    EXIT TOE, T0S % exit T
A1: % end present
    GOTO A0 % end loop
TOE: % end trap;
```

≡

```
AWAIT S % await S
```

Example:

`abort` can translate to

```
ABORT % most general case
LABORT % if no other [L]ABORTS are included in abort scope
TABORT % if neither || nor other [L|T]ABORTS are included
```

Furthermore: valued signals, pre delay expressions, ...
The KEP Instruction Set

- Includes all kernel statements
- In addition, some derived statements

This redundancy improves space/time efficiency

<table>
<thead>
<tr>
<th>TOS:</th>
<th>% trap T in</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0:</td>
<td>% loop</td>
</tr>
<tr>
<td>PAUSE</td>
<td>% pause;</td>
</tr>
<tr>
<td>PRESENT S,A1</td>
<td>% present S then</td>
</tr>
<tr>
<td>EXIT TOE, TOS</td>
<td>% exit T</td>
</tr>
<tr>
<td>A1:</td>
<td>% end present</td>
</tr>
<tr>
<td>GOTO A0</td>
<td>% end loop</td>
</tr>
<tr>
<td>TOE:</td>
<td>% end trap;</td>
</tr>
</tbody>
</table>

- Refined ISA to reduce HW usage

Example: abort can translate to

ABORT in the most general case
LABORT if no other [L]ABORTS are included in abort scope
TABORT if neither || nor other [L|T]ABORTS are included
The KEP Instruction Set

- Includes all kernel statements
- In addition, some derived statements

This redundancy improves space/time efficiency

<table>
<thead>
<tr>
<th>TOS:</th>
<th>% trap T in</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0:</td>
<td>% loop</td>
</tr>
<tr>
<td>PAUSE</td>
<td>% pause;</td>
</tr>
<tr>
<td>PRESENT S, A1</td>
<td>% present S then</td>
</tr>
<tr>
<td>EXIT TOE, TOS</td>
<td>% exit T</td>
</tr>
<tr>
<td>A1:</td>
<td>% end present</td>
</tr>
<tr>
<td>GOTO A0</td>
<td>% end loop</td>
</tr>
<tr>
<td>TOE:</td>
<td>% end trap;</td>
</tr>
</tbody>
</table>

≡

AWAIT S % await S

- Refined ISA to reduce HW usage
  Example: abort can translate to
  
  ABORT in the most general case
  
  LABORT if no other [L]ABORTS are included in abort scope
  
  TABORT if neither || nor other [L|T]ABORTS are included

- Furthermore: valued signals, pre, delay expressions, ...
The Kiel Esterel Processor Architecture

- Reactive Core
  - Decoder & Controller, Reactive Block, Thread Block
- Interface Block
  - Interface signals, Local signals, ...
- Data Handling
  - Register file, ALU, ...
The Architecture of the Reactive Core
Overview

Introduction

The Kiel Esterel Processor

The Compiler
  The Concurrent KEP Assembler Graph (CKAG)
  Handling Thread Dependencies

Wrap-Up
The KEP Compiler

Thread scheduling:
1. Construct **Concurrent KEP Assembler Graph (CKAG)**
2. Compute thread priorities/ids that respect dependencies
3. Generate PAR and PRI0 statements accordingly
The KEP Compiler

Thread scheduling:
1. Construct **Concurrent KEP Assembler Graph (CKAG)**
2. Compute thread priorities/\textit{ids} that respect dependencies
3. Generate PAR and PRI0 statements accordingly

Other tasks:
- Analyze Watcher requirements
- Map Esterel statements to KEP refined ISA
The KEP Compiler

Thread scheduling:
1. Construct **Concurrent KEP Assembler Graph (CKAG)**
2. Compute thread priorities/ids that respect dependencies
3. Generate PAR and PRI0 statements accordingly

Other tasks:
- Analyze Watcher requirements
- Map Esterel statements to KEP refined ISA

Optimizations:
- Dead code elimination, based on CKAG
- “Undismantling” of kernel statements
module EXAMPLE:
output A,R;
end module

module EXAMPLE

% module Example
OUTPUT A,R
module EXAMPLE:
output A,R;
[
  |
  |
]
end module

% module Example
OUTPUT A,R
module EXAMPLE:
output A,R;
[
|
]
end module

% module Example
OUTPUT A,R
module EXAMPLE:
  output A,R;
 |
 ||
 |
 end module

% module Example
 OUTPUT A,R

L01: PAR 1,A0,1
L02: PAR 1,A1,2
L03: PARE A2
L04: A0:
L08: A1:
module EXAMPLE:
output A,R;
[  
  weak abort
  when A
  ||
]
end module

% module Example
  OUTPUT A,R
L01: PAR 1,A0,1
L02: PAR 1,A1,2
L03: PARE A2
L04: A0:
L08: A1:
module EXAMPLE:
output A,R;
[ weak abort
  when A ||
]
end module

% module Example
OUTPUT A,R
L01: PAR 1,A0,1
L02: PAR 1,A1,2
L03: PARE A2
L04: A0:
L08: A1:
L04 WABORT A,A3

module EXAMPLE
L03 fork
L04 A0 L08 A1
1

module EXAMPLE:
output A,R;
[
  weak abort
when A
||
]
end module

---

\% module Example
OUTPUT A,R

L01: PAR 1,A0,1
L02: PAR 1,A1,2
L03: PARE A2
L04: A0: WABORT A,A3
L08: A3:A1:
module EXAMPLE:
output A,R;
[
    weak abort
    sustain R
    when A
]
]
end module

% module Example
OUTPUT A,R
L01: PAR 1,A0,1
L02: PAR 1,A1,2
L03: PARE A2
L04: A0: WABORT A,A3
L08: A3:A1:
module EXAMPLE:
output A,R;
[
    weak abort
    sustain R
    when A
||
]
end module

---

% module Example
OUTPUT A,R

L01: PAR 1,A0,1
L02: PAR 1,A1,2
L03: PARE A2
L04: A0: WABORT A,A3
L05: A4
L05 EMIT R
L06: PAUSE
L07: GOTO A4
L08: A3:A1:
module EXAMPLE:
output A,R;
[
  weak abort
  sustain R
  when A
]
end module

% module Example
OUTPUT A,R

L01: PAR 1,A0,1
L02: PAR 1,A1,2
L03: PARE A2
L04: A0: WABORT A,A3
L08: A3:A1:

L05: EMIT R
L06: PAUSE
L07: GOTO A4
L08: A3:A1:
module EXAMPLE:
output A,R;
[ weak abort sustain R when A ]
end module

% module Example
OUTPUT A,R
L01: PAR 1,A0,1
L02: PAR 1,A1,2
L03: PARE A2
L04: A0: WABORT A,A3
L05: A4: EMIT R
L06: PAUSE
L07: GOTO A4
L08: A3:A1:
module EXAMPLE:
output A,R;
[
  weak abort
  sustain R
  when A
||
  pause;
]
end module

% module Example
OUTPUT A,R
L01: PAR 1,A0,1
L02: PAR 1,A1,2
L03: PARE A2
L04: A0: WABORT A,A3
L05: A4: EMIT R
L06: PAUSE
L07: GOTO A4
L08: A3:A1:

```
module EXAMPLE:
output A,R;
[
  weak abort
  sustain R
  when A
||
  pause;
]
end module
```
module EXAMPLE:
output A,R;
[
  weak abort
  sustain R
  when A
  ||
  pause;
]
end module

% module Example
OUTPUT A,R

L01: PAR 1,A0,1
L02: PAR 1,A1,2
L03: PARE A2
L04: A0: WABORT A,A3
L05: A4: EMIT R
L06: PAUSE
L07: GOTO A4
L08: A3:A1:
module EXAMPLE:
output A,R;
[
  weak abort
  sustain R
when A
||
pause;
]
end module

% module Example
OUTPUT A,R

L01: PAR 1,A0,1
L02: PAR 1,A1,2
L03: PARE A2
L04: A0: WABORT A,A3
L05: A4: EMIT R
L06: PAUSE
L07: GOTO A4
L08: A3:A1:PAUSE
module EXAMPLE:
output A,R;
[
  weak abort
  sustain R
  when A
||
  pause;
  present R then
  emit A
end present
]
end module

% module Example
OUTPUT A,R

L01:  PAR 1,A0,1
L02:  PAR 1,A1,2
L03:  PARE A2
L04:  A0:  WABORT A,A3
L05:  A4:  EMIT R
L06:  PAUSE
L07:  GOTO A4
L08:  A3:A1:PAUSE
module EXAMPLE:
output A,R;
[
  weak abort
  sustain R
  when A
  ||
  pause;
  present R then emit A
  end present
]
end module

% module Example
OUTPUT A,R

L01: PAR 1,A0,1
L02: PAR 1,A1,2
L03: PARE A2
L04: A0: WABORT A,A3
L05: A4: EMIT R
L06: PAUSE
L07: GOTO A4
L08: A3:A1:PAUSE
L09: PRESENT R,A5
L10: EMIT A
L11: A5
module EXAMPLE:
output A,R;
[
  weak abort
  sustain R
  when A
]
||
pause;
present R then emit A
end present
] end module
module EXAMPLE:
output A,R;
[  
  weak abort
  sustain R
  when A
  ||
  pause;
  present R then
  emit A
  end present
]
end module

% module Example
OUTPUT A,R

L01: PAR 1,A0,1
L02: PAR 1,A1,2
L03: PARE A2
L04: A0: WABORT A,A3
L05: A4: EMIT R
L06: PAUSE
L07: GOTO A4
L08: A3:A1:PAUSE
L09: PRESENT R,A5
L10: EMIT A
L11: A5:
module EXAMPLE:
output A, R;
[ weak abort sustain R when A || pause;
present R then emit A end present ] end module

% module Example
OUTPUT A, R

L01: PAR 1, A0, 1
L02: PAR 1, A1, 2
L03: PARE A2
L04: A0: WABORT A, A3
L05: A4: EMIT R
L06: PAUSE
L07: GOTO A4
L08: A3: A1: PAUSE
L09: PRESENT R, A5
L10: EMIT A
L11: A5: A2: JOIN
L12: HALT
Dealing With Dependencies

1. Add dependencies to CKAG
Dealing With Dependencies

1. Add dependencies to CKAG
2. Compute statement priorities
Dealing With Dependencies

1. Add dependencies to CKAG
2. Compute statement priorities
3. Assign initial thread prios
Dealing With Dependencies

1. Add dependencies to CKAG
2. Compute statement priorities
3. Assign initial thread prios
4. Insert PRI0 stmts

module: EXAMPLE
[T0] PAR*
[T2] A0
 2
[T1] A1
 1
[T2] WABORT A,A3
[T2] A4
[T2] EMIT R
[T2] PRESENT R,A5
[T2] GOTO A4
[T0] JOIN 0
[T0] HALT

⇒

module: EXAMPLE
[T0,P2] PAR*
[T2,P2] A0
 2
[T1,P1] A1
 1
[T2,P2] WABORT A,A3
[T2,P2] A4
[T2,P2] EMIT R
[T2,P1] PRESENT R,A5
[T2,P1/2] PAUSE
[T1,P2] PRIO 2
[T2,P2] GOTO A4
[T0,P1/1] HALT

[Xin Li, Marian Boldt, Reinhard v. Hanxleden Multi-Threaded Reactive Processing]
Dealing With Dependencies

% module Example

OUTPUT A,R
EMIT _TICKLEN,#14

[L01,T0,P2] PAR 2,A0,2
[L02,T0,P2] PAR 1,A1,1
[L03,T0,P2] PARE A2,2
[L04,T2,P2] A0: WABORT A,A3
[L05,T2,P2] A4: EMIT R
[L06,T2,P1] PRIO 1
[L07,T2,P2] PRIO 2
[L08,T2,P1/2] PAUSE
[L09,T2,P2] GOTO A4
[L10,T2,P1] A3: NOTHING
[L11,T1,P1] A1: PRIO 2
[L12,T1,P1/2] PAUSE
[L13,T1,P2] PRESENT R,A5
[L14,T1,P2] EMIT A
[L15,T1,P1] A5: PRIO 1
[L16,T1,P1] NOTHING
[L17] A2: JOIN 0
[L18,T0,P1/1] HALT
Example—Execution Trace

Scheduling criteria: 1. active, 2. highest priority, 3. highest id

module EXAMPLE:
output A,R;
[
  weak abort
  sustain R;
  when A
  |
  pause;
  present R then
  emit A
end present
]
end module
CKAG for tcint
The Concurrent KEP Assembler Graph (CKAG)

Handling Thread Dependencies

tcint ThreadId’s
tcint ThreadId’s with Thread Dependencies
Overview

Introduction

The Kiel Esterel Processor

The Compiler

Wrap-Up

KEP Evaluation Platform
Experimental Results
Summary & Outlook
Identify that benchmark...
The KEP Evaluation Platform

- Highly automated process, currently using 470+ benchmarks
- End to end validation of hardware and compiler against “trusted” reference (Esterel Studio)
- Detailed performance measurements
## Code Characteristics and Compilation Times

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Esterel Threads</th>
<th>Preemptions</th>
<th>KEP3a (Unoptimized)</th>
<th>Preemption handled by</th>
<th>Compiling Time (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cnt</td>
<td>Max Depth</td>
<td>Conc</td>
<td>Cnt</td>
<td>Max Depth</td>
</tr>
<tr>
<td>abcd</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>abcdef</td>
<td>6</td>
<td>2</td>
<td>6</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>eight_but</td>
<td>8</td>
<td>2</td>
<td>8</td>
<td>40</td>
<td>2</td>
</tr>
<tr>
<td>chan_prot</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>reactor_ctrl</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>runner</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>example</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>ww_button</td>
<td>13</td>
<td>3</td>
<td>4</td>
<td>27</td>
<td>2</td>
</tr>
<tr>
<td>greycounter</td>
<td>17</td>
<td>3</td>
<td>13</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>tcint</td>
<td>39</td>
<td>5</td>
<td>17</td>
<td>18</td>
<td>2</td>
</tr>
<tr>
<td>mca200</td>
<td>59</td>
<td>5</td>
<td>49</td>
<td>64</td>
<td>4</td>
</tr>
</tbody>
</table>

### Note:
In the mca200, the watcher refinement reduces the hardware requirements from 4033 slices (if all preemptions were handled by general purpose Watchers) to 3265 slices (19% reduction).
## Worst-/Average-Case Reaction Times

<table>
<thead>
<tr>
<th>Module Name</th>
<th>MicroBlaze</th>
<th>KEP3a-Unoptimized</th>
<th>KEP3a-Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WCRT</td>
<td>ACRT</td>
<td>WCRT Ratio to</td>
</tr>
<tr>
<td></td>
<td>V5</td>
<td>V7</td>
<td>CEC</td>
</tr>
<tr>
<td>abc</td>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abc</td>
<td>d</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>eight</td>
<td>but</td>
<td></td>
<td></td>
</tr>
<tr>
<td>chan</td>
<td>prot</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reactor</td>
<td>ctrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>runner</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>example</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ww</td>
<td>button</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grey</td>
<td>counter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tcint</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mca</td>
<td>200</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The worst-/average-case reaction times, in clock cycles, for the KEP3a and MicroBlaze:

- **WCRT speedup**: typically >4x
- **ACRT speedup**: typically >5x
- **Optimizations yield further improvements**
### Memory Usage

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Esterel LOC</th>
<th>MicroBlaze Code+Data (byte)</th>
<th>KEP3a-Unopt. Code (word)</th>
<th>KEP3a-opt. Code (word)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>V5</td>
<td>V7</td>
<td>CEC</td>
<td>abs. rel.</td>
</tr>
<tr>
<td>abcd</td>
<td>160</td>
<td>6680</td>
<td>7928</td>
<td>7212</td>
</tr>
<tr>
<td>abcdef</td>
<td>236</td>
<td>9352</td>
<td>9624</td>
<td>9220</td>
</tr>
<tr>
<td>eight_but</td>
<td>312</td>
<td>12016</td>
<td>11276</td>
<td>11948</td>
</tr>
<tr>
<td>chan_prot</td>
<td>42</td>
<td>3808</td>
<td>6204</td>
<td>3364</td>
</tr>
<tr>
<td>reactor_ctrl</td>
<td>27</td>
<td>2668</td>
<td>5504</td>
<td>2460</td>
</tr>
<tr>
<td>runner</td>
<td>31</td>
<td>3140</td>
<td>5940</td>
<td>2824</td>
</tr>
<tr>
<td>example</td>
<td>20</td>
<td>2480</td>
<td>5196</td>
<td>2344</td>
</tr>
<tr>
<td>ww_button</td>
<td>76</td>
<td>6112</td>
<td>7384</td>
<td>5980</td>
</tr>
<tr>
<td>greycounter</td>
<td>143</td>
<td>7612</td>
<td>7936</td>
<td>8688</td>
</tr>
<tr>
<td>tcint</td>
<td>355</td>
<td>14860</td>
<td>11376</td>
<td>15340</td>
</tr>
<tr>
<td>mca200</td>
<td>3090</td>
<td>104536</td>
<td>77112</td>
<td>52998</td>
</tr>
</tbody>
</table>

- **Unoptimized:** 83% avg reduction of memory usage (Code+RAM)
- **Optimized:** May yield further 5% to 30+% improvements
Power Consumption

<table>
<thead>
<tr>
<th>Module Name</th>
<th>MicroBlaze (82mW@50MHz)</th>
<th>KEP3a $^1$ (mW)</th>
<th>Ratio (KEP to MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Idle</td>
<td>Peak</td>
<td>Idle</td>
</tr>
<tr>
<td>abcd</td>
<td>69</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>abcdef</td>
<td>74</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>eight_but</td>
<td>74</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>chan_prot</td>
<td>70</td>
<td>28</td>
<td>12</td>
</tr>
<tr>
<td>reactor_ctrl</td>
<td>76</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>runner</td>
<td>78</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>example</td>
<td>77</td>
<td>25</td>
<td>9</td>
</tr>
<tr>
<td>ww_button</td>
<td>81</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>greycounter</td>
<td>78</td>
<td>44</td>
<td>33</td>
</tr>
<tr>
<td>tcint</td>
<td>80</td>
<td>18</td>
<td>10</td>
</tr>
</tbody>
</table>

- Peak energy usage reduction: 75% avg
- Idle (= no inputs) energy usage reduction: 86% avg

$^1$Based on Xilinx 3S200-4ft256, requires an additional 37mW as quiescent power for the chip itself
## Scalability

### Synthesis results for Xilinx 3S1500-4fg-676²

<table>
<thead>
<tr>
<th>Thread Count</th>
<th>Slices</th>
<th>Gates (k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1295</td>
<td>295</td>
</tr>
<tr>
<td>10</td>
<td>1566</td>
<td>299</td>
</tr>
<tr>
<td>20</td>
<td>1871</td>
<td>311</td>
</tr>
<tr>
<td>40</td>
<td>2369</td>
<td>328</td>
</tr>
<tr>
<td>60</td>
<td>3235</td>
<td>346</td>
</tr>
<tr>
<td>80</td>
<td>4035</td>
<td>373</td>
</tr>
<tr>
<td>100</td>
<td>4569</td>
<td>389</td>
</tr>
<tr>
<td>120</td>
<td>5233</td>
<td>406</td>
</tr>
</tbody>
</table>

- 48 valued signals<br>  up to 256 possible
- 2 Watchers, 8 Local Watchers<br>  either up to 64 possible
- 1k (1024) instruction words<br>  up to 64k possible
- 128 registers (in word)<br>  up to 512 possible
- 16-bits (65536) max counter value
- Frequency is stable (around 60 MHz)

²For comparison, a MicroBlaze implementation requires around 1k slices and 309k gates; a two threads EMPEROR platform requires around 2k slices
## Analysis of Context Switches

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Instr's total abs.</th>
<th>CSs total abs.</th>
<th>CSs at same priority abs.</th>
<th>PRI0s total abs.</th>
<th>CSs due to PRI0 abs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcdd</td>
<td>16513</td>
<td>3787</td>
<td>1521</td>
<td>3082</td>
<td>1243</td>
</tr>
<tr>
<td>abcd</td>
<td>29531</td>
<td>7246</td>
<td>3302</td>
<td>6043</td>
<td>2519</td>
</tr>
<tr>
<td>eight_but</td>
<td>39048</td>
<td>10073</td>
<td>5356</td>
<td>8292</td>
<td>3698</td>
</tr>
<tr>
<td>chan_prot</td>
<td>5119</td>
<td>1740</td>
<td>707</td>
<td>990</td>
<td>438</td>
</tr>
<tr>
<td>reactor_ctrl</td>
<td>151</td>
<td>48</td>
<td>29</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>runner</td>
<td>5052</td>
<td>704</td>
<td>307</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>example</td>
<td>208</td>
<td>60</td>
<td>2</td>
<td>26</td>
<td>9</td>
</tr>
<tr>
<td>ww_button</td>
<td>292</td>
<td>156</td>
<td>92</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>greycounter</td>
<td>160052</td>
<td>34560</td>
<td>14043</td>
<td>26507</td>
<td>12725</td>
</tr>
<tr>
<td>tcint</td>
<td>80689</td>
<td>33610</td>
<td>16769</td>
<td>5116</td>
<td>2129</td>
</tr>
<tr>
<td>mca200</td>
<td>982417</td>
<td>256988</td>
<td>125055</td>
<td>242457</td>
<td>105258</td>
</tr>
</tbody>
</table>
Summary Reactive Processors

Processor supports reactive control flow directly, at hardware level

- “Watchers” monitor preemption signals
  *No need for polling, interrupts*

- Support for concurrency
  *Multi-threading or multi-processing*

- Synchronous model of computation
  *Perfectly deterministic, predictable timing*
Outlook

- Improve priority assignments
Outlook

- Improve priority assignments
- Speedup signal expression computations with external logic block
Outlook

- Improve priority assignments
- Speedup signal expression computations with external logic block
- WCRT analysis with concurrency
Outlook

- Improve priority assignments
- Speedup signal expression computations with external logic block
- WCRT analysis with concurrency
- Extend to Esterel v7
Outlook

- Improve priority assignments
- Speedup signal expression computations with external logic block
- WCRT analysis with concurrency
- Extend to Esterel v7
- KEP in Esterel—e.g., to produce Esterel virtual machine
Outlook

- Improve priority assignments
- Speedup signal expression computations with external logic block
- WCRT analysis with concurrency
- Extend to Esterel v7
- KEP in Esterel—e.g., to produce Esterel virtual machine
- Combination with multi-core (e.g., for data handling)
Outlook

- Improve priority assignments
- Speedup signal expression computations with external logic block
- WCRT analysis with concurrency
- Extend to Esterel v7
- KEP in Esterel—e.g., to produce Esterel virtual machine
- Combination with multi-core (e.g., for data handling)
- Adaptation to non-Esterel languages
Outlook

- Improve priority assignments
- Speedup signal expression computations with external logic block
- WCRT analysis with concurrency
- Extend to Esterel v7
- KEP in Esterel—e.g., to produce Esterel virtual machine
- Combination with multi-core (e.g., for data handling)
- Adaptation to non-Esterel languages
- Apply compilation approach to standard processors ("multi-threaded simulation")
Outlook

- Improve priority assignments
- Speedup signal expression computations with external logic block
- WCRT analysis with concurrency
- Extend to Esterel v7
- KEP in Esterel—e.g., to produce Esterel virtual machine
- Combination with multi-core (e.g., for data handling)
- Adaptation to non-Esterel languages
- Apply compilation approach to standard processors (“multi-threaded simulation”)
- To go further:
  http://www.informatik.uni-kiel.de/rtsys/kep/
Identify that benchmark . . .
Identify that benchmark . . .

Thanks!
Questions/Comments?
Overview

KEP3a Instruction Set + Architecture
  Esterel-Type Instructions
  Handling Concurrency
  Handling Preemption
  Handling Exceptions

The Compiler

Further Measurements

Summary
### Instruction Set Summary 1/2

<table>
<thead>
<tr>
<th>Mnemonic, Operands</th>
<th>Esterel Syntax</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PAR Prio, startAddr [, ID]</strong></td>
<td>`[ p</td>
<td></td>
</tr>
<tr>
<td><strong>PARE endAddr</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>JOIN</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PRIO Prio</strong></td>
<td></td>
<td>Set the priority of the current thread</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic, Operands</th>
<th>Esterel Syntax</th>
<th>Notes</th>
</tr>
</thead>
</table>
| **[W]ABORT [n,] S, endAddr** | `[weak] abort ... when [n] S` | `$S$ can be one of the following:  
1. $S$: signal status (present/absent)  
2. PRE($S$): previous status of signal  
3. TICK: always present  
n can be one of the following:  
1. $#data$: immediate data  
2. `reg`: register contents  
3. `?S`: value of a signal  
4. PRE(`?S`): previous value of a signal |
| **[W]ABORTI S, endAddr** | `[weak] abort ... when immediate $S$` | |
| **SUSPEND[I] S, endAddr** | `suspend ... when [immediate] $S$` | |
| **EXIT TrapEnd[, TrapStart]** | `trap $T$ in  
exit $T$  
end trap` | Exit from a trap, TrapStart and TrapEnd specify trap scope. Unlike GOTO, check for concurrent EXITs and terminate enclosing `||`
## Instruction Set Summary 2/2

<table>
<thead>
<tr>
<th>Mnemonic, Operands</th>
<th>Esterel Syntax</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAUSE</td>
<td>pause</td>
<td>Wait for a signal. AWAIT TICK is equivalent to PAUSE</td>
</tr>
<tr>
<td>AWAIT ([n,] S)</td>
<td>await ([n] S)</td>
<td></td>
</tr>
<tr>
<td>AWAIT[I] (S)</td>
<td>await ([\text{immediate}] S)</td>
<td></td>
</tr>
<tr>
<td>CAWAITS</td>
<td>await</td>
<td></td>
</tr>
<tr>
<td>CAWAITS[I] (S, \text{addr})</td>
<td>case ([\text{immediate}]) (S) do end</td>
<td>wait for several signals in parallel</td>
</tr>
<tr>
<td>CAWAITE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGNAL (S)</td>
<td>signal (S) in ...end</td>
<td>Initialize a local signal (S)</td>
</tr>
<tr>
<td>EMIT (S, {#\text{data}</td>
<td>\text{reg}})</td>
<td>emit (S[\text{val}])</td>
</tr>
<tr>
<td>SUSTAIN (S, {#\text{data}</td>
<td>\text{reg}})</td>
<td>sustain (S[\text{val}])</td>
</tr>
<tr>
<td>PRESENT (S, \text{elseAddr})</td>
<td>present (S) then ...end</td>
<td>Jump to (\text{elseAddr}) if (S) is absent</td>
</tr>
<tr>
<td>NOTHING</td>
<td>nothing</td>
<td>Do nothing</td>
</tr>
<tr>
<td>HALT</td>
<td>halt</td>
<td>Halt the program</td>
</tr>
<tr>
<td>GOTO (\text{addr})</td>
<td>loop ...end loop</td>
<td>Jump to (\text{addr})</td>
</tr>
<tr>
<td>CALL (\text{addr})</td>
<td>call (P)</td>
<td>call a procedure, and return from the procedure</td>
</tr>
<tr>
<td>RETURN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Handling Concurrency

Execution status of a single thread

The status of the whole program, as managed by the Thread Block
Handling Concurrency

A thread has its

- **thread id**
- **address range** and independent **program counter**
- **priority value**
  - assigned when a thread is created
  - dynamically changed via PRI0 instruction
- **status flags**
  - ThreadEnable
  - ThreadActive

---

```plaintext
% Esterel
[  
  p  
  | |  
  q ];
```

```plaintext
% KEP Assembler
PAR 1,A0,1
PAR 1,A1,2
PARE A2
A0: p
A1: q
A2: JOIN
```
Handling Preemption

Watcher contains

Enable Watcher (EW)
   ▶ Watches the PC (Program Counter)
   ▶ Compares PC
   ▶ Preemption enabled?

Trigger Watcher (TW)
   ▶ Watches the Signal
   ▶ Counts down the counter (abortion)
   ▶ Preemption active?

% Esterel
abort
   weak abort
   p;
   when S2;
   q;
   when S1;

% KEP Assembler
ABORT S1,A1
WABORT S2,A0
p
A0: q
A1:
Watcher Refinement

**Thread Watcher**  
- belongs to a thread directly  
- can neither include concurrent threads nor other preemptions  
- least powerful, but also cheapest

**Local Watcher**  
- may include concurrent threads and also preemptions handled by a Thread Watcher  
- cannot include another Local Watcher

**Watcher**  
- may include concurrent threads and any preemptions  
- most powerful, but also most expensive
Handling Exceptions

Exception

- has its address range
- sets an exitFlag
  - cleared when reaching the end of the trap scope
  - effects control at the join point
- can be overridden based on the corresponding trap scopes (address range)
Overview

KEP3a Instruction Set + Architecture

The Compiler
The Concurrent KEP Assembler Graph
Kep Thread Ids
Priority Assigning
Cyclicity

Further Measurements

Summary
Esterel Kep Compiling Scheme

- parsing and module expanding is done by Columbia Esterel Compiler (CEC)
- result is an Esterel abstract syntax tree (AST)
- dismantle some more complex esterel statements to simple statements
- creating assembler program and Concurrent KEP Assembler Graph (CKAG) by recursive visiting of the AST
- do computations and optimizations on CKAG
  - e.g. do priority assigning to ensure correct behavior
  - e.g. do statement collapsing to use less statements
- printing to KEP assembler or viewing the CKAG
Concurrent KEP Assembler Graph (CKAG)

- represents the control flow of KEP assembler
- needed to realize more complex compiling computations
- each node contains a KEP statement and *Thread Id*
- different kind of nodes
  - instantaneous: TransientNode, LabelNode
  - not instantaneous: DelayNode
  - parallel: ForkNode, JoinNode
- different kind of edges
  - sequence control flow
  - abort preemption, weak abort preemption,
  - exit preemption
Kep Thread Id

- contains an unique integer value and the parent’s thread pointer
- the main thread has no parent thread (NULL pointer)
- a sequence of fork nodes declares the sub-thread id’s
- Kep Thread Id structure has much similarity to a tree
- three important relations between thread id’s
  - subthread
  - concurrent
  - sequence
Thread Id Example 1

- T1 and T2 are sub-threads of T0
- T1 and T2 are concurrent to each other
- no sequence behaviour
Thread Id Example 2

- \{T1,\ldots,T7\} are sub-threads of T0
- T4 and T5 are sub-threads of T3
- \{T1,T2,T3\}, \{T1,T2,T4,T5\} and \{T6,T7\} are in each case concurrent to each other
- \{T1,T2,T3,T4,T5\} are in sequence to \{T6,T7\}
Priority Assigning Scheme

- determining signal dependencies
- relevant for our concern are only concurrent signal dependencies, which we call priority dependencies
- run priority assigning algorithm fulfilling all priority constraints
- if assigning algorithm ends successfully:
  - setting priorities of PAR and PARE statements
  - inserting priority statements according to the priority assigning if necessary
Signal Dependencies

- a writer-reader pair of the same signal $S$ defines a signal dependency

- writer statements: EMIT, SUSTAIN, SIGNAL, EXIT, SETV

- reader statements: PRESENT, AWAIT, PAUSE, HALT, SUSTAIN, EMIT
Signal Dependency Example 1

```
EMIT S

PRESENT S,A0
```
Signal Dependency Example 2

```plaintext
SETV S,#2
EMIT T,?S
PRESENT T,A1
```
Priority Dependencies

- A priority dependency is defined as signal dependency with concurrent \textit{KepThreadIds}.
- Execution order can and must be determined by priorities respectively thread id’s.
- Priority dependencies are only relevant when executable within the same tick, otherwise superfluous.
- Analysis is conservative in a way that superfluous dependencies may exist.
Priority Dependency Example

```
PAR*

...  

EMIT S

...  

PRESENT S,A1

...  

PRESENT S,A0
```
Constraints

- a priority dependency \((w, r)\) causes a writer constraint:
  \[
  w.prio >_w r.prio := \\
  w.prio > r.prio \lor (w.prio = r.prio \land w.id > r.id)
  \]
- instantaneous execution order causes control flow constraints: priorities never grow instantaneously
- node: the control flow successors of DelayNodes causes no constraints because they does not represent instantaneous execution
Assigning Algorithm

- try to fulfill all constraints by a DFS like algorithm
- basic idea to compute the priority \( n.prio \) of a node \( n \):

\[
n.prio = \max(\{n.inst\_succ.prio\} \cup \{n.reader.prio + 1\})
\]

- for a DelayNode \( d \) additionally the priority \( d.prio\_next \) of the next instant is computed: this ensures termination of a one pass traversal

\[
d.prio\_next = \max(\{n.next\_succ.prio\} \cup \{n.reader.prio + 1\})
\]
Priority Dismantling

- sometimes we cannot fulfill the constraints because of statements which are writer and reader at the same time:
  - SUSTAIN S
  - EMIT S, expr
- separate writing and reading part by dismantling
- priority statements can be inserted in between
The Concurrent KEP Assembler Graph

PAR*

WABORTI A,A3

SUSTAIN R

AWAIT R

EMIT A

JOIN 0

Xin Li, Marian Boldt, Reinhard v. Hanxleden
The Concurrent KEP Assembler Graph

Kep Thread Ids

Priority Assigning

Cyclicity

PAR*

WABORTI A,A3

AWAIT R

EMIT R

PRIO 1

AWAIT R

PRIO 2

EMIT A

PAUSE

GOTO A4

JOIN 0

PAUSE

PRIO 1

EMIT A

PAR*

A4

A3

...
PAR*

EMIT V1, ?V2

PRESENT V1, A3

EMIT V2, #3

A3
PAR*
EMIT V1
PRESENT V1,A3
SETV V1,?V2
EMIT V2,#3
PAR

EMIT V1

PRESENT V1,A3

EMIT V2,#3

SETV V1,?V2

A3
Thread Id Optimization

- same priority:
  - thread id has scheduling relevance
  - no priority statements needed
  - try to optimize thread id’s

- thread dependency: priority dependency modulo nodes/statements

- optimization: assign dependency fathers higher thread id then children

- node: sub-thread constraints must be obtained
Superfluous Priority Dependencies

- Priority dependencies which are not executable within the same instant are called **superfluous** dependencies
- in general difficult to compute
- special case:
  - search least common fork node (nodes are concurrent)
  - make instant analysis starting with fork node
  - if one node has only *instantaneous* and the other *not instantaneous* paths: remove Priority Dependency
The Concurrent KEP Assembler Graph

PAR*

- EMIT S
- PAUSE
- PRESENT S, A0

Xin Li, Marian Boldt, Reinhard v. Hanxleden Multi-Threaded Reactive Processing Slide 71
PAR*

EMIT S

PAUSE

PRESENT S, A0
CKAG Node Types

The CKAG distinguishes the following sets of nodes:

**D**: Delay nodes (octagons)
- PAUSE, AWAIT, HALT, SUSTAIN

**F**: Fork nodes (triangles)
- PAR/PARE

**T**: Transient nodes (rectangles/inverted triangles)
- EMIT, PRESENT, etc. (rectangles)
- JOIN nodes (inverted triangles)

**N**: Set of all nodes, $N = D \cup F \cup T$
The Concurrent KEP Assembler Graph (CKAG)

Define

- for each fork node $n$:
  - $n$.join: the JOIN statement corresponding to $n$,
  - $n$.sub: the transitive closure of nodes in threads generated by $n$.

- for abort nodes $n$ ($[\mathsf{L|T}]\mathsf{[W]}\mathsf{ABORT[I]}, \mathsf{SUSPEND[I]}$):
  - $n$.end: the end of the abort scope opened by $n$,
  - $n$.scope: the nodes within $n$’s abort scope.

- for all nodes $n$:
  - $n$.prio: the priority that the thread executing $n$ should be running with

- for $n \in D \cup F$,
  - $n$.prionext: the priority that the thread executing $n$ should be resumed with in the subsequent tick.
CKAG Dependency Types

Define dependencies

\( \text{n.dep}_i \): the dependency sinks with respect to \( n \) at the current tick (the *immediate dependencies*)

\( \text{n.dep}_d \): the dependency sinks with respect to \( n \) at the next tick (the *delayed dependencies*)

Induced by emissions of strong abort trigger signals and corresponding delay nodes within the abort scope
CKAG Successor Types

Define following types of successors for each $n$:

- $\text{n.suc}_c$: the *control successors*.
- $\text{n.suc}_w$: the *weak abort successors*
- $\text{n.suc}_s$: the *strong abort successors*
- $\text{n.suc}_f$: the *flow successors*

  the set $\text{n.suc}_c \cup \text{n.suc}_w \cup \text{n.suc}_s$

For $n \in F$ we also define the following *fork abort successors*

- $\text{n.suc}_{wf}$: the *weak fork abort successors*
- $\text{n.suc}_{sf}$: the *strong fork abort successors*
Program Cycle

An Esterel program is considered cyclic iff the corresponding CKAG contains a path from a node to itself, where for all nodes $n$ and their successors along that path, $n'$ and $n''$, the following holds:

$$
n \in D \land n' \in n.suc_w
\lor n \in F \land n' \in n.suc_c \cup n.suc_{wf}
\lor n \in T \land n' \in n.suc_c \cup n.dep_i
\lor n \in T \land n' \in n.dep_d \land n'' \in n'.suc_c \cup n'.suc_s \cup n'.suc_{sf}.
$$
Overview

KEP3a Instruction Set + Architecture

The Compiler

Further Measurements

- Code Characteristics and Compilation Times
- Speed, Size, Power, Scalability
- Analysis of context switches
- Another Example

Summary
### Code Characteristics and Compilation Times

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Esterel</th>
<th>KEP3a (Unoptimized/optimized)</th>
<th>MicroBlaze (V5/V7/CEC)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Threads</td>
<td>Preemptions</td>
<td>Compiling Time (Sec)</td>
</tr>
<tr>
<td></td>
<td>Cnt Max</td>
<td>Depth Conc</td>
<td>CKAG</td>
</tr>
<tr>
<td></td>
<td>Nodes Max</td>
<td>Depth</td>
<td>PRIO</td>
</tr>
<tr>
<td></td>
<td>Depth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abc</td>
<td>4 2 4</td>
<td>20 2</td>
<td>211 36 3 30</td>
</tr>
<tr>
<td>abcd</td>
<td>6 2 6</td>
<td>30 2</td>
<td>313 90 3 48</td>
</tr>
<tr>
<td>eight_but</td>
<td>8 2 8</td>
<td>40 2</td>
<td>415 168 3 66</td>
</tr>
<tr>
<td>chan_prot</td>
<td>5 3 4</td>
<td>6 1</td>
<td>80 4 2 10</td>
</tr>
<tr>
<td>reactor_ctrl</td>
<td>3 2 3</td>
<td>5 1</td>
<td>51 5 1 0</td>
</tr>
<tr>
<td>runner</td>
<td>2 2 2</td>
<td>9 3</td>
<td>61 0 1 0</td>
</tr>
<tr>
<td>example</td>
<td>2 2 2</td>
<td>4 2</td>
<td>36 2 3 6</td>
</tr>
<tr>
<td>ww_button</td>
<td>13 3 4</td>
<td>27 2</td>
<td>194 0 1 0</td>
</tr>
<tr>
<td>greycounter</td>
<td>17 3 13</td>
<td>19 2</td>
<td>414 53 6 58</td>
</tr>
<tr>
<td>tcint</td>
<td>39 5 17</td>
<td>18 2</td>
<td>583 65 3 20</td>
</tr>
<tr>
<td>mca200</td>
<td>59 5 49</td>
<td>64 4</td>
<td>11219 129 11 190</td>
</tr>
</tbody>
</table>

**Note:** In the mca200, the watcher refinement reduces the hardware requirements from 4033 slices (if all preemptions were handled by general purpose Watchers) to 3265 slices (19% reduction).
Worst-/Average-Case Reaction Times

<table>
<thead>
<tr>
<th>Module Name</th>
<th>MicroBlaze WCRT</th>
<th>MicroBlaze ACRT</th>
<th>KEP3a-Unoptimized WCRT Ratio to best MB</th>
<th>KEP3a-Unoptimized ACRT Ratio to best MB</th>
<th>KEP3a-optimized WCRT Ratio to Unopt</th>
<th>KEP3a-optimized ACRT Ratio to Unopt</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcd</td>
<td>1559</td>
<td>1464</td>
<td>135</td>
<td>87</td>
<td>135</td>
<td>84</td>
</tr>
<tr>
<td>abcdef</td>
<td>2281</td>
<td>2155</td>
<td>201</td>
<td>120</td>
<td>201</td>
<td>117</td>
</tr>
<tr>
<td>eight_but</td>
<td>3001</td>
<td>2833</td>
<td>267</td>
<td>159</td>
<td>267</td>
<td>153</td>
</tr>
<tr>
<td>chan_prot</td>
<td>754</td>
<td>683</td>
<td>117</td>
<td>60</td>
<td>117</td>
<td>54</td>
</tr>
<tr>
<td>reactor_ctrl</td>
<td>487</td>
<td>456</td>
<td>54</td>
<td>45</td>
<td>51</td>
<td>39</td>
</tr>
<tr>
<td>runner</td>
<td>566</td>
<td>512</td>
<td>36</td>
<td>15</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>example</td>
<td>467</td>
<td>404</td>
<td>42</td>
<td>24</td>
<td>42</td>
<td>24</td>
</tr>
<tr>
<td>ww_button</td>
<td>1185</td>
<td>1148</td>
<td>72</td>
<td>51</td>
<td>48</td>
<td>36</td>
</tr>
<tr>
<td>greycounter</td>
<td>1965</td>
<td>1851</td>
<td>528</td>
<td>375</td>
<td>528</td>
<td>375</td>
</tr>
<tr>
<td>tcint</td>
<td>3580</td>
<td>3488</td>
<td>408</td>
<td>252</td>
<td>342</td>
<td>204</td>
</tr>
<tr>
<td>mca200</td>
<td>75488</td>
<td>73824</td>
<td>2862</td>
<td>1107</td>
<td>2862</td>
<td>1107</td>
</tr>
</tbody>
</table>

The worst-/average-case reaction times, in clock cycles, for the KEP3a and MicroBlaze.

- WCRT speedup: typically >4x
- ACRT speedup: typically >5x
- Optimizations yield further improvements
## Memory Usage

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Esterel LOC</th>
<th>MicroBlaze Code+Data (byte)</th>
<th>KEP3a-Unopt. Code (word)</th>
<th>KEP3a-opt. Code (word)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>V5</td>
<td>V7</td>
<td>CEC</td>
</tr>
<tr>
<td>abcde</td>
<td>160</td>
<td>6680</td>
<td>7928</td>
<td>7212</td>
</tr>
<tr>
<td>abcdef</td>
<td>236</td>
<td>9352</td>
<td>9624</td>
<td>9220</td>
</tr>
<tr>
<td>eight but</td>
<td>312</td>
<td>12016</td>
<td>11276</td>
<td>11948</td>
</tr>
<tr>
<td>chan_prot</td>
<td>42</td>
<td>3808</td>
<td>6204</td>
<td>3364</td>
</tr>
<tr>
<td>reactor_ctrl</td>
<td>27</td>
<td>2668</td>
<td>5504</td>
<td>2460</td>
</tr>
<tr>
<td>runner</td>
<td>31</td>
<td>3140</td>
<td>5940</td>
<td>2824</td>
</tr>
<tr>
<td>example</td>
<td>20</td>
<td>2480</td>
<td>5196</td>
<td>2344</td>
</tr>
<tr>
<td>ww_button</td>
<td>76</td>
<td>6112</td>
<td>7384</td>
<td>5980</td>
</tr>
<tr>
<td>greycounter</td>
<td>143</td>
<td>7612</td>
<td>7936</td>
<td>8688</td>
</tr>
<tr>
<td>tcint</td>
<td>355</td>
<td>14860</td>
<td>11376</td>
<td>15340</td>
</tr>
<tr>
<td>mca200</td>
<td>3090</td>
<td>104536</td>
<td>77112</td>
<td>52998</td>
</tr>
</tbody>
</table>

- **Unoptimized:** 83% avg reduction of memory usage (Code+RAM)
- **Optimized:** May yield further 5% to 30+% improvements
## Power Consumption

<table>
<thead>
<tr>
<th>Module Name</th>
<th>MicroBlaze (82mW@50MHz)</th>
<th>KEP3a&lt;sup&gt;3&lt;/sup&gt; (mW)</th>
<th>Ratio (KEP to MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Idle</td>
<td>Peak</td>
<td>Idle</td>
</tr>
<tr>
<td>abcd</td>
<td>69</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>abcdef</td>
<td>74</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>eight_but</td>
<td>74</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>chan_prot</td>
<td>70</td>
<td>28</td>
<td>12</td>
</tr>
<tr>
<td>reactor_ctrl</td>
<td>76</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>runner</td>
<td>78</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>example</td>
<td>77</td>
<td>25</td>
<td>9</td>
</tr>
<tr>
<td>ww_button</td>
<td>81</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>greycounter</td>
<td>78</td>
<td>44</td>
<td>33</td>
</tr>
<tr>
<td>tcint</td>
<td>80</td>
<td>18</td>
<td>10</td>
</tr>
</tbody>
</table>

- Peak energy usage reduction: 75% avg
- Idle (= no inputs) energy usage reduction: 86% avg

<sup>3</sup>Based on Xilinx 3S200-4ft256, requires an additional 37mW as quiescent power for the chip itself.
### Scalability

Synthesis results for Xilinx 3S1500-4fg-676

<table>
<thead>
<tr>
<th>Thread Count</th>
<th>Slices</th>
<th>Gates (k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1295</td>
<td>295</td>
</tr>
<tr>
<td>10</td>
<td>1566</td>
<td>299</td>
</tr>
<tr>
<td>20</td>
<td>1871</td>
<td>311</td>
</tr>
<tr>
<td>40</td>
<td>2369</td>
<td>328</td>
</tr>
<tr>
<td>60</td>
<td>3235</td>
<td>346</td>
</tr>
<tr>
<td>80</td>
<td>4035</td>
<td>373</td>
</tr>
<tr>
<td>100</td>
<td>4569</td>
<td>389</td>
</tr>
<tr>
<td>120</td>
<td>5233</td>
<td>406</td>
</tr>
</tbody>
</table>

- 48 valued signals  
  up to 256 possible
- 2 Watchers, 8 Local Watchers  
  either up to 64 possible
- 1k (1024) instruction words  
  up to 64k possible
- 128 registers (in word)  
  up to 512 possible
- 16-bits (65536) max counter value
- Frequency is stable (around 60 MHz)

---

4For comparison, a MicroBlaze implementation requires around 1k slices and 309k gates; a two threads EMPEROR platform requires around 2k slices.
## Analysis of Context Switches

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Instr's total abs.</th>
<th>CSs total abs.</th>
<th>CSs at same priority abs.</th>
<th>PRI0s total abs.</th>
<th>CSs due to PRI0 abs.</th>
<th>abs. ratio</th>
<th>abs. rel.</th>
<th>abs. rel. rel.</th>
<th>abs. rel. rel.</th>
<th>abs. rel. rel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcd</td>
<td>16513</td>
<td>3787</td>
<td>1521</td>
<td>3082</td>
<td>1243</td>
<td>4.36</td>
<td>0.40</td>
<td>0.33</td>
<td>0.40</td>
<td></td>
</tr>
<tr>
<td>abcdef</td>
<td>29531</td>
<td>7246</td>
<td>3302</td>
<td>6043</td>
<td>2519</td>
<td>4.08</td>
<td>0.46</td>
<td>0.35</td>
<td>0.42</td>
<td></td>
</tr>
<tr>
<td>eight_but</td>
<td>39048</td>
<td>10073</td>
<td>5356</td>
<td>8292</td>
<td>3698</td>
<td>3.88</td>
<td>0.53</td>
<td>0.37</td>
<td>0.45</td>
<td></td>
</tr>
<tr>
<td>chan_prot</td>
<td>5119</td>
<td>1740</td>
<td>707</td>
<td>990</td>
<td>438</td>
<td>2.94</td>
<td>0.41</td>
<td>0.25</td>
<td>0.44</td>
<td></td>
</tr>
<tr>
<td>reactor_ctrl</td>
<td>151</td>
<td>48</td>
<td>29</td>
<td>0</td>
<td>0</td>
<td>3.15</td>
<td>0.60</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>runner</td>
<td>5052</td>
<td>704</td>
<td>307</td>
<td>0</td>
<td>0</td>
<td>7.18</td>
<td>0.44</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>example</td>
<td>208</td>
<td>60</td>
<td>2</td>
<td>26</td>
<td>9</td>
<td>3.47</td>
<td>0.30</td>
<td>0.15</td>
<td>0.35</td>
<td></td>
</tr>
<tr>
<td>ww_button</td>
<td>292</td>
<td>156</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1.87</td>
<td>0.59</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>greycounter</td>
<td>160052</td>
<td>34560</td>
<td>14043</td>
<td>26507</td>
<td>12725</td>
<td>4.63</td>
<td>0.41</td>
<td>0.37</td>
<td>0.48</td>
<td></td>
</tr>
<tr>
<td>tcint</td>
<td>80689</td>
<td>33610</td>
<td>16769</td>
<td>5116</td>
<td>2129</td>
<td>2.4</td>
<td>0.50</td>
<td>0.06</td>
<td>0.42</td>
<td></td>
</tr>
<tr>
<td>mca200</td>
<td>982417</td>
<td>256988</td>
<td>125055</td>
<td>242457</td>
<td>105258</td>
<td>3.82</td>
<td>0.49</td>
<td>0.41</td>
<td>0.43</td>
<td></td>
</tr>
</tbody>
</table>
Edwards02: Esterel to KEP

```
module Edwards02:
  input S, I;
  output O;

  signal A,R in
    every S do
      p
      end
  end signal

  await S;
  loop
    abort
    p;
    halt
    when S
    end loop

  sustain S
  loop
    emit S;
    pause;
    end loop
  end loop

  every S do
    p
  end

  loop
    p;
  end loop

  A:
    p;
    goto A
end module
```
Edwards02: a Possible Execution Trace

module Edwards02:
input S, I;
output O;
signal A,R in
  every S do
  await I;
  weak abort sustain R;
  when immediate A;
  emit O;
||
  loop
  pause;
  pause;
present R then
  emit A;
end present
end every
end signal
end module
module Edwards02:
input S, I;
output 0;
signal A,R in
every S do
    await I;
    weak abort
    sustain R;
when immediate A;
emit 0;
||
loop
pause;
pause;
present R then
emit A;
end present
end every
end signal
end module
Overview

KEP3a Instruction Set + Architecture

The Compiler

Further Measurements

Summary

Multi-processing vs. Multi-threading
Comparison of Synthesis Options
Application Scenarios
Multi-processing vs. Multi-threading

Multi-processing (EMPEROR/RePIC)

- Esterel thread \(\approx\) one independent RePIC processor
- Thread Control Unit handles the synchronization and communication
- Three-valued signal representation
- \texttt{sync} command to synchronize threads

Multi-threading (KEP)

- Esterel thread \(\approx\) several registers
- priority-based scheduler
- \texttt{PRI0} command to synchronize threads
## Comparison of Synthesis Options

<table>
<thead>
<tr>
<th></th>
<th>HW</th>
<th>SW</th>
<th>Co-design</th>
<th>Reactive Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multi-processing</td>
<td>Multi-threading</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Speed</strong></td>
<td>++</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Flexibility</strong></td>
<td>-</td>
<td>++</td>
<td>-</td>
<td>+/-</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>+</td>
<td>++</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td><strong>Logic Area</strong></td>
<td>++/−</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>++</td>
<td>−</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td><strong>Power Usage</strong></td>
<td>++</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td><strong>Appl. Design Cycle</strong></td>
<td>−</td>
<td>++</td>
<td>+/−</td>
<td>++</td>
</tr>
</tbody>
</table>
Scenario I: DSP + Reactive Processor

- Global Memory
- Communication Backplane
- DSP
- Reactive Processor
- IPs
Scenario II: DSP + HW Block + Reactive Processor
Scenario III: HW Block + Reactive Processor
Possible Co-Design Development Flow

Reactive processing . . .

▸ permits a simple mapping strategy
▸ allows optimizations on high-level
▸ can meet stricter constraints than classical architectures
▸ permits a better tradeoff between all cost factors

Thanks/Comments?