A Behavioral Synthesis Frontend
to the Haste/TiDE design flow

Sune F. Nielsen
Jens Sparsø
Jonas B. Jensen
Johan S. R. Nielsen
Introduction / Motivation

- Most high-level async tools are based on syntax-directed translation (Haste/TiDE, Balsa, ...)
- One-to-one mapping from source program to implementation. Handshake components = syntactic elements in source code.
- No automatic optimization. Designer has to implement alternative solutions.
- Optimized code very hard to understand.
- Very different from behavioural synthesis in synchronous design: specification + constraints/goals for the desired implementation.

- Idea underlying this work: adapt existing (synchronous) behavioural synthesis techniques to the domain of asynchronous design.
Outline

1. Motivation, background, and contributions.
2. Overview of the design flow.
3. Related work.
4. CDFG representation of a source program
5. Data-path implementation
6. Haste code generation
7. Results
8. Conclusion
The Haste/TiDE design flow
This work – an extension to The Haste/TiDE design flow

- Target architecture (controller+datapath)
- Data-path components (latch, FF, ALU, MUL, etc.)

IEEE Async'09

A Behavioral Synthesis Frontend to the Haste/TiDE design flow

DTU Informatics, Technical University of Denmark

5
This work – an extension to The Haste/TiDE design flow

- Target architecture (controller+datapath)
- Data-path components (latch, FF, ALU, MUL, etc.)

- One-to-one mapping used to advantage.
- Direct control of synthesized implementation at high level
- Avoids tedious back-end design
Previous work at DTU


Targeted the Balsa system from U. of Manchester.

Demonstrated the *idea*.

Shortcomings:

• Assumed the existence of a CDFG-representation of the code to be synthesized. CDFG -> Synthesis -> Balsa program.

• Limited set of CDFG’s: A loop with DFG-body. No procedures and functions
Contributions of the current work

A fully automatic, Haste-in-Haste-out synthesis tool, which can handle large non-trivial code examples.

1. The tool now supports the Haste language.
2. Front-end: Source code -> CDFG representation.
3. Much larger set of source language constructs, with arbitrary compositions of loops, procedures etc.
4. Haste allow certain circuit level optimizations not available in Balsa.
5. Haste/TiDE tool flow provides reliable (i.e. realistic) area and power figures:
   - Cost functions in optimization
   - Benchmark results
Contributions of the current work

A **fully automatic, Haste-in-Haste-out synthesis tool, which can handle large non-trivial code examples.**

1. The tool now supports the Haste language.
2. Front-end: Source code -> CDFG representation.
3. Much larger set of constructs, with arbitrary compositions of loops, procedures etc.
4. Haste allows certain circuit level optimizations not available in Balsa.
5. Haste/TiDE tool flow provides reliable (i.e. realistic) area and power figures:
   - Cost functions in optimization
   - Benchmark results
Flow overview
(Haste code example)

& word = type [0..2^16-1]
& EX: main proc(X0,X1,X2?chan word & Y0,Y1!chan word).
beg
  & a0 = const 255
  & a1 = const 255
  & a2 = const 255
  & a3 = const 255
  & x0,x1,x2,y0,y1: var word
  | forever do
    X0?x0 || X1?x1 || X2?x2 ;
    y0 := ((a0+x0)+(x0*x1)-a1) fit word ;
    if x1>a2 then
      y1:= (a3*(x1+x2)) fit word
    else
      y1:= (x1-x2) fit word
    fi ;
    Y0!y0 || Y1!y1
  od
end
Flow overview
(CDFG for loop body)

\[
\begin{align*}
X0 \oplus x0 & \quad \lor \quad X1 \oplus x1 & \quad \lor \quad X2 \oplus x2 ; \\
y0 & := ((a0+x0)+(x0\times x1)−a1) \text{ fit word} ; \\
\text{if } x1 > a2 & \text{ then} \\
& \quad \quad y1 := (a3 \times (x1+x2)) \text{ fit word} \\
\text{else} & \quad \quad y1 := (x1−x2) \text{ fit word} \\
& \quad \text{fi} ; \\
Y0 \oplus y0 & \quad \lor \quad Y1 \oplus y1
\end{align*}
\]
Flow overview
(Generic data-path template)
Flow overview
(Haste implementation of generic datapath)
Flow overview
( Behavioral synthesis)

• **Scheduling:**
  – Fine grain discrete-time model.
  – Operator nodes of CDFG placed into (one or more) time-slots

• **Allocation:**
  – Determine (minimum) required hardware resources: functional units, registers/variables, multiplexing, and control.

• **Binding:**
  – Operator nodes and (temporary) variables are mapped onto specific hardware resources: latches/FFs and FU’s.

• Optimization using simulated annealing (cost function = area or speed)
• Solution space reduced by ASAP and ALAP schedules.
Scheduling, assignment and binding
Related work


Source code to CDFG

- More elaborate than what is hinted in most literature. Precise definitions are hard to find.
- Fundamentally a CDFG is a 1-bounded Petri net.

\[
X0?x0 \ || \ X1?x1 \ || \ X2?x2 ~
\]
\[
y0 := ((a0+x0)+(x0*x1)-a1) \text{ fit word}
\]
\[
\text{if } x1>a2 \text{ then}
\]
\[
y1 := (a3*(x1+x2)) \text{ fit word}
\]
\[
\text{else}
\]
\[
y1 := (x1-x2) \text{ fit word}
\]
\[
\text{fi ;}
\]
\[
Y0!y0 \ || \ Y1!y1
\]
Source code to CDFG

- More elaborate than what is hinted in most literature. Precise definitions are hard to find.
- Fundamentally a CDFG is a 1-bounded Petri net.
- Source code and CDFG must have same black-box behaviour.

```
X0?x0 || X1?x1 || X2?x2 ;
y0 := ((a0+x0)+(x0*x1)−a1) fit word
if x1>a2 then
    y1:=(a3*(x1+x2)) fit word
else
    y1:=(x1−x2) fit word
fi ;
Y0!y0 || Y1!y1
```
Source code to CDFG

- More elaborate than what is hinted in most literature. Precise definitions are hard to find.
- Fundamentally a CDFG is a 1-bounded Petri net.
- Source code and CDFG must have same black-box behaviour.

\[
\begin{align*}
X_0?x_0 & || X_1?x_1 & || X_2?x_2 ; \\
y_0 := ((a_0+x_0)+(x_0*x_1)−a_1) \text{ fit word} \\
\text{if } x_1>a_2 \text{ then} \\
y_1 := (a_3*(x_1+x_2)) \text{ fit word} \\
\text{else} \\
y_1 := (x_1−x_2) \text{ fit word} \\
fi ; \\
Y_0!y_0 & || Y_1!y_1
\end{align*}
\]
Source code to CDFG

- More elaborate than what is hinted in most literature. Precise definitions are hard to find.
- Fundamentally a CDFG is a 1-bounded Petri net.
- Source code and CDFG must have same black-box behaviour.

\[ X_0 ? x_0 || X_1 ? x_1 || X_2 ? x_2 ; \]
\[ y_0 := \text{if } x_1 > a_2 \text{ then } \]
\[ y_1 := (a_3*(x_1+x_2)) \text{ fit word} \]
\[ \text{else } \]
\[ y_1 := (x_1-x_2) \text{ fit word} \]
\[ \text{fi ;} \]
\[ Y_0!y_0 || Y_1!y_1 \]
Source code to CDFG

- More elaborate than what is hinted in most literature. Precise definitions are hard to find.
- Fundamentally a CDFG is a 1-bounded Petri net.
- Source code and CDFG must have same black-box behaviour.

\[
\begin{align*}
X0 \cdot x0 & \lor X1 \cdot x1 \lor X2 \cdot x2 ; \\
y0 & := ((a0 + x0) + (x0 \cdot x1) - a1) \text{ fit word} \\
\text{if } x1 > a2 \text{ then} \\
y1 & := (a3 \cdot (x1 + x2)) \text{ fit word} \\
\text{else} \\
y1 & := (x1 - x2) \text{ fit word} \\
\text{fi} ; \\
Y0 \cdot y0 & \lor Y1 \cdot y1
\end{align*}
\]
Complete CDFG

... including forever-do loop
CDFG nodes for Send and Receive
CDFG nodes for Send and Receive
CDFG nodes for Send and Receive
CDFG nodes for Send and Receive
CDFG nodes for procedure call
Implementation and optimizations

Variables
• Latches and flip-flops

Functional Units
• Chaining
• Multiple pull-type output channels. Same FU computes different results: $a<b$, $a-b$, etc

Functions and procedures:
• Input muxes can be avoided when input parameters are the same (variables), by defining a parameterless function.
Functional Unit Templates

Basic FU  
Chaining  
Multiple (alternative) pull-type outputs  
Negation enables swapping inputs  
Combining FU templates
Synthesized Haste code

& P0 : main proc (c4?chan [0..(2^16-1)])
  & c3?chan [0..(2^16-1)]
  & c2?chan [0..(2^16-1)]
  & c1?chan [0..(2^16-1)]
  & c0?chan [0..(2^16-1)]).

begin
  & v0: var [0..(2^16-1)]
  & v1: var [0..(2^16-1)]
  & v2: var [0..(2^16-1)] ff
  & v3: var [0..(2^16-1)] ff

  & MUL0 : func(a,b?var [0..(2^16-1)]): [0..(2^16-1)] . a*b
  & ALU0 : func(a,b?var [0..(2^16-1)]): [0..(2^16-1)] . a+b

  & ALU1: func(a,b?var [0..(2^16-1)]):
    [[bool,[0..(2^16-1)]]].begin
    & sub  : func():[0..(2^17-1)]. (a-b)
    & fit  [0..(2^17-1)]
    & subb = alias sub cast [[bool,[0..(2^16-1)]]
    & subb.0] subb.0] end

FU " MUL1" w. two output's declared
- subb.0 difference (i.e., a-b)
- subb.1 is sign (i.e., a>b)
Synthesized Haste code

```haste
| do (1) cast bool then
| c4?v0 || c3?v1 || c2?v2 |
v3 := ALU0(a0,v0) ;
v3 := ALU0(v3, MUL0(v0,v1)) ;
v3 := ALU1(v3,a1).0 ;
if (~ALU1(v1,a2).1) then
  v2 := MUL0(a3,ALU0(v1,v2))
else
  v2 := ALU1(v1,v2).0
fi ;
c1?v3 || c0?v2
od
end
```

ALU0 used to compute $v3 := a0 + v0$

ALU0 chained with MUL0
$v3 := v3 + v1 \times v2$

ALU1 used to compute $v3 := v3 - a1$

if $v1 > a2$ then ...
$[v1 - a2; \text{if not bit}_16 \text{ then ...}]$
### Results – optimizing for area

<table>
<thead>
<tr>
<th>Benchmark Program</th>
<th>Source code</th>
<th>Synthesized code (Area)</th>
<th>Relative Area [%]</th>
<th>Relative Delay [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ALU #</td>
<td>Mul #</td>
<td>Var #</td>
<td>Area [$\mu m^2$]</td>
</tr>
<tr>
<td>GCD</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2586</td>
</tr>
<tr>
<td>FIR4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>11759</td>
</tr>
<tr>
<td>FIR8</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>25831</td>
</tr>
<tr>
<td>FIR16</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>56519</td>
</tr>
<tr>
<td>FIR8COEF</td>
<td>7</td>
<td>8</td>
<td>33</td>
<td>55756</td>
</tr>
<tr>
<td>HAL</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>15897</td>
</tr>
<tr>
<td>SEVENTH</td>
<td>15</td>
<td>15</td>
<td>16</td>
<td>46189</td>
</tr>
<tr>
<td>ELLIPTIC</td>
<td>26</td>
<td>8</td>
<td>26</td>
<td>50127</td>
</tr>
<tr>
<td>COSINE</td>
<td>26</td>
<td>16</td>
<td>32</td>
<td>69104</td>
</tr>
<tr>
<td>FBANK</td>
<td>34</td>
<td>24</td>
<td>48</td>
<td>96902</td>
</tr>
<tr>
<td>QUAD</td>
<td>17</td>
<td>6</td>
<td>24</td>
<td>39165</td>
</tr>
<tr>
<td>JPEG</td>
<td>29</td>
<td>3</td>
<td>21</td>
<td>63964</td>
</tr>
</tbody>
</table>

Area: 5-58% reduction (avg. 30%)
Results – optimizing for speed

<table>
<thead>
<tr>
<th>Benchmark Program</th>
<th>Source code</th>
<th>Synthesized code (Speed)</th>
<th>Rel. Area [%]</th>
<th>Rel. Delay [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ALU #</td>
<td>Mul #</td>
<td>Var #</td>
<td>Area [µm²]</td>
</tr>
<tr>
<td>GCD</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2586</td>
</tr>
<tr>
<td>FIR4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>11759</td>
</tr>
<tr>
<td>FIR8</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>25831</td>
</tr>
<tr>
<td>FIR16</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>56519</td>
</tr>
<tr>
<td>FIR8COEF</td>
<td>7</td>
<td>8</td>
<td>33</td>
<td>55756</td>
</tr>
<tr>
<td>HAL</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>15897</td>
</tr>
<tr>
<td>SEVENTH</td>
<td>15</td>
<td>15</td>
<td>16</td>
<td>46189</td>
</tr>
<tr>
<td>ELLIPTIC</td>
<td>26</td>
<td>8</td>
<td>26</td>
<td>50127</td>
</tr>
<tr>
<td>COSINE</td>
<td>26</td>
<td>16</td>
<td>32</td>
<td>69104</td>
</tr>
<tr>
<td>FBANK</td>
<td>34</td>
<td>24</td>
<td>48</td>
<td>96902</td>
</tr>
<tr>
<td>QUAD</td>
<td>17</td>
<td>6</td>
<td>24</td>
<td>39165</td>
</tr>
<tr>
<td>JPEG</td>
<td>29</td>
<td>3</td>
<td>21</td>
<td>63964</td>
</tr>
</tbody>
</table>

Speed: Up to 67% reduction (avg. 40%)
Conclusion

• A fully automatic Haste-in-Haste-out synthesis tool.
• The tool can handle large non-trivial subset of Haste.
• Results:
  – Area: 5-58% reduction (avg. 30%)
  – Speed: 0-67% reduction (avg. 40%)

• Source-to-source optimization (behavioural synthesis) combined with syntax-directed-translation is a promising approach.
• Using syntax-directed-translation as backend for a synthesis system from <your favourite high level language> seems promising as well.
References