Semi-Automatic Derivation of aiT Timing Models from HW Specifications

Markus Pister  Marc Schlickling

Compiler Design Lab  AbsInt GmbH
Computer Science Dept.  Science Park 1
Saarland University  Saarbrücken

Verisoft XT project meeting – 11.12.2007
Outline

• Motivation

• Semi-automatic pipeline analyzer generation
  ◦ The approach
  ◦ Hardware specifications
  ◦ Processing the VHDL model
  ◦ Analysis framework
  ◦ Abstractions on VHDL models
  ◦ Code generation

• Improvements to current methods

• Conclusion
Motivation

- Hard real time scenarios makes computer aided validation of safety critical embedded systems crucial
- Computation of WCET is a key issue in validation of safety critical applications
Motivation

- aiT Timing Analyzer
  - Based on abstract interpretation
  - WCET estimation mainly based on *pipeline analysis* modelling the processor pipeline and system controllers
  - Today: Pipeline models are hand-crafted
    → time consuming and error-prone process

- Modern processors are derived from formal hardware descriptions

- Why not derive the pipeline analysis from the hardware description of a processor?
Motivation

Problems

- [Availability/Accessibility of hardware specification]
- Processor specification too large to be used in aiT
- Specification needs to be abstracted

Idea

- Use of static methods to derive an abstracted model that is suitable for use in aiT
Overview of the derivation process

Variable \( z \) is a \text{std_logic_vector}(0\text{ to }1);

\begin{align*}
\text{case } x \text{ is} \\
\text{when } "00" \Rightarrow \ldots \\
\text{when } "01" \Rightarrow \ldots \\
\text{when others} \Rightarrow \ldots \\
\end{align*}

Variable \( irq \) is a \text{std_logic_vector}(0\text{ to }1);

\begin{align*}
\text{case } x \text{ is} \\
\text{when } "00" \Rightarrow \ldots \\
\text{when } "01" \Rightarrow \ldots \\
\text{when others} \Rightarrow \ldots \\
\end{align*}

Variable \( x \) is a \text{std_logic_vector}(0\text{ to }1);

\begin{align*}
\text{case } x \text{ is} \\
\text{when } "00" \Rightarrow \ldots \\
\text{when } "01" \Rightarrow \ldots \\
\text{when others} \Rightarrow \ldots \\
\end{align*}

\text{Static Analyses/ Abstractions}

\text{Code Generation}

\text{VHDL Specification}

\text{VHDL2Crl2}

\text{Timing Analyser}

\text{C-Code}

\text{Abstracted VHDL}

\text{Objective}

\text{Legend}

\text{Completed action}

\text{Ongoing work}

\text{Objective}
Hardware description language
- Hierarchically organized
- Defined in the IEEE standard 1076
- Focus on
  - register-transfer-level (RTL)
  - synthesizable IEEE substandard 1076.6

```vhdl
entity counter is
  port(clk:in std_logic; rst:in std_logic;
       val:out std_logic_vector(2 downto 0));
end;

architecture rtl of counter is
  signal cnt:std_logic_vector(2 downto 0);
begin
  P1: process(clk,rst) is
    if (rst='1') then
      cnt<="000";
    elsif (rising_edge(clk)) then
      cnt<=cnt+'1';
    end if;
  end;
  P2: process(cnt) is
    val<=cnt;
  end;
end;
```
VHDL semantics

Two-level semantics

- Process execution
- Synchronization + Restart + Time

Process execution

- Sequential, imperative semantics
- Assignments to signals are delayed
- Executes, until suspended (by wait statement)

Second level

- After all processes have suspended
- Check if restart of processes is necessary
  - Yes: restart these processes (delta cycle)
  - No: wait for timeout (theta cycle)
- Repeat
Transformed semantics

Ordering of process execution is not important

- Variables are local
- Signal assignments take effect only at synchronization point

Transform two-level semantics to one level

- Always execute all processes in fixed ordered loop
- Signal assignments can be viewed as assignments to new variables (copied at synchronization point)
  - assignment: $s \leftarrow '1'; \implies s_{new} := '1';$
  - at sync: $s := s_{new};$
- Add a guard to process header to check, if re-execution in the next loop iteration is necessary
  - Guard true, iff process is restarted at synchronization of previous iteration
VHDL2CrL2

- Parses the fully synthesizable VHDL substandard
- Elaborates the parsed entity forest according to IEEE standard.
- Emits a semantically equivalent CrL2 description of the VHDL model.
  - by expressing VHDL as sequential program
- Generates some help constructs to support static analyses
  - e.g. specific routines for the analysis start, clock modelling and environmental signals.
Control-Flow Representation Language (Crl2)

- Provides a textual description of a combined call and control flow graph
- Hierarchically organized in
  - Routines,
  - Basic blocks,
  - Instructions and
  - Edges
- Extensible using an attribute-value concept
Overview of the derivation process

Variable \( z \) is a \( \text{Std_logic_vector(0 to 1)} \);

```
case x is
  when "00" => …
  when "01" => …
  when others => …
end case;
```

Variable \( \text{irq} \) is a \( \text{std_logic_vector(0 to 1)} \);

```
case x is
  when "00" => …
  when "01" => …
  when others => …
end case;
```

Variable \( x \) is a \( \text{std_logic_vector(0 to 1)} \);

```
case x is
  when "00" => …
  when "01" => …
  when others => …
end case;
```

Static Analyses/Abstractions

```
variable work_package_x is
  std_logic_vector(0 to 1);
if work_package_x = "00"
  then A
elsif x = "01"
  then B
else C
endif;
```

C-Code

```
std_logic work_package_x(2);
if(work_package_x[0]==0
  && work_package_x[1]==0)
  A
else
  if(work_package_x[0]==0
    && work_package_x[1]==1)
    B
  else
    C
  endif;
```

Legend

- Completed action
- Ongoing work
- Objective

Objective

VerisoftXT - 11.12.2007  Markus Pister, Marc Schlickling  Derivation of aiT Timing Models
Analysis framework

- VHDL model is transformed into CRL2 description
  - semantically equivalent
  - some syntactical modifications (e.g. switch statements are transformed into if-then-else statements)
- Analyzer based on PAG specification
- PAG generates code working on CRL2 description
Sequential execution of processes modelled by `simul` routine

- "Process execution" is guarded by the `simul_if` modelling the sensitivity list of the process
- Analyzer decides, whether the edge to the call has to be taken or not

Synchronization point is represented by `simul_wait`

`environment` routine allows analysis of open designs
• Clock events have to be modelled separately
• Introduced special *clock* routine signalizing rising or falling events via special attributes
• Suppress uninteresting events, e.g. Leon 2 SPARC V8 implementation completely triggered on rising clock edges
• Support for multiple clock domains
Abstractions on VHDL models

1. Dead-Code Elimination
   - Slice all parts being unreachable under a specified assumption
   - Decreases the size of the model

2. Process Substitution
   - Replace a process with an abstract process
     - Semantic of the abstract process specified in an arbitrary language (e.g. C)
     - Changing of domains necessary
       - Transforming data types (e.g. addresses to address intervals)

3. Memory Abstraction
   - Remove the memory from the VHDL model
   - Introduce new interface
     - Necessary to insert instructions into the model
     - Can be done by inserting abstract processes
Abstractions for generation of timing analysis

- Memory abstraction
- Find constraint: ”When does an instruction leave the pipeline?”
  - Identify point in the model, where instructions complete
  - After passing this point, the completed instruction does not have any effect on signals, etc.
- Compute a backward slice for this constraint
  - All parts being not part of this slice have no effect on the timing
Overview of the derivation process

Variable $z$ is a `std_logic_vector(0 to 1);

```
case x is
  when "00" => …
  when "01" => …
  when others => …
end case;
```

Variable $irq$ is also a `std_logic_vector(0 to 1);

```
case x is
  when "00" => …
  when "01" => …
  when others => …
end case;
```

Variable $x$ is another `std_logic_vector(0 to 1);

```
case x is
  when "00" => …
  when "01" => …
  when others => …
end case;
```

Abstracted VHDL

Variable work_package_x is a `std_logic_vector(0 to 1);

```
if work_package_x = "00"
  then A
elsif x = "01"
  then B
else C
endif;
```

C-Code

```
std_logic work_package_x[2];
if(work_package_x[0]==0
  && work_package_x[1]==0){
  A
} else {
  if(work_package_x[0]==0
    && work_package_x[1]==1){
    B
  } else {
    C
  }
}
```

Legend

- Completed action
- Ongoing work
- Objective
Abstracted VHDL

- Reconstructed from transformed $\text{CRL2}$.
- Hierarchy preserving.
- May be used for further analyses (model checking, etc.)

C-Code

- Abstract simulation of (abstracted) VHDL for exactly one processor step (single-cycle update)
- Suitable for usage in the aiT tool chain, i.e. the cache/pipeline analysis.
Generating an aiT timing analysis

Iterate until model is handable

- Generate C-code for the model
- Simulate the resulting model (using aiT)
  - Check for state explosions and
  - Check state differences
- Substitute a process with an abstract one
  - e.g. cache abstraction
- Eliminate dead code
Improvements to current methods

- Less error prone by reducing the human involvement
- Precise information about processor behavior from VHDL specification
- Much faster than manual implementation (months vs. days/weeks)
- Better adaption to faster development cycles in industry

Industrial/academical partners

University of Dortmund  
Airbus France  
Verisoft XT  
ES_PASS  
AVACS  
Infineon
Conclusion

- WCET is one of the key issues for validating safety-critical embedded systems
- Computation of safe upper bounds on the WCET using aiT.
- Manual development of a pipeline analysis as component within aiT
  - hand crafted,
  - error prone and
  - time consuming.
- Idea of semi-automatically generation of pipeline analysers
- Practical benefit of this method over existing one