Outline

1 Motivation

2 Timing Model Derivation

3 Analysis Framework

4 Usability

5 Conclusion
Motivation

- Embedded systems supporting our daily life
- Safety-critical systems often have to fulfill strict timing constraints to ensure a proper functioning
- Guaranteeing the timeliness of these systems is of crucial importance (and also required by Certification Authorities)
The Timing Problem

- Runtime of a task varies between
  - different inputs
  - and different runs

- Measuring the WCET of a task is impossible on complex architectures

- Static methods derive upper bounds on the WCET independently from concrete inputs
The Timing Problem

- Runtime of a task varies between
  - different inputs
  - and different runs

- Measuring the WCET of a task is impossible on complex architectures

- Static methods derive upper bounds on the WCET independently from concrete inputs
The Timing Problem

Runtime of a task varies between
- different inputs
- and different runs

Measuring the WCET of a task is impossible on complex architectures

Static methods derive upper bounds on the WCET independently from concrete inputs
Static WCET Analysis Framework

- Implemented in the aiT tool
- Based on reconstructed control flow
- Cache/pipeline analysis models instruction flow through the processor
  - Relies on timing model of underlying processor
  - Abstract simulation of task execution
Modern Processor Development

- Modern processors are highly configurable and offer advanced features like
  - Caches and deep pipelines
  - Out-of-order execution
  - Speculation and branch prediction

- Timing models must reflect timing behavior of the hardware

- Processors designed using formal hardware description languages (HDLs)

- HDLs are explicitly designed to support
  - Design
  - Simulation and
  - Verification

- Timing behavior already part of the specification
Timing Model Derivation Methodology

- Model preprocessing eliminates parts not relevant for the timing behavior
- Processor state abstraction approximates parts of the model
- Static analysis techniques useful to support
  - Model preprocessing, and
  - Model understanding
- Semantics of HDLs special compared to “normal” programming languages
  - Abstract semantics that enables use of program analyses
1 Process execution

- Sequential, imperative semantics
- Assignments to variables immediately take effect
- Assignments to signals are delayed
- Executes, until suspended

2 Process reactivation

- After all processes have suspended
- Check if restart of processes is necessary
  - Yes: restart these processes (delta cycle)
  - No: wait for timeout/external signal change

VHDL model $\equiv$ set of processes $p_l$, with $p_l = (\zeta_l, \Pi_l, \omega_l)$, and $l \in \mathbb{L}$
Transformed Semantics

- Ordering of process execution is not important
  - Variables are process-local
  - Signal assignments take effect only at synchronization point

- Transform two-level semantics to one level
  - Signal assignments can be viewed as assignments to new variable
    - Assignment \( s \leftarrow V : \Theta[s] \leftarrow V \)
    - At sync: \( \forall s \in \text{Signals} : \Theta[s] \leftarrow \Theta(s), s \leftarrow \Theta(s) \)
  - Always execute all processes in fixed ordered loop
  - Add guard controlling the reexecution of process \( p_l \)
    - Guard true, iff \( \Theta \vdash \bigvee_{s \in \omega_l} (s \neq s) \)

Level-reduction transforms data dependency between processes into control dependency
Abstract Semantics

- Processes directly mapped to control-flow graph
  - Statements map to single nodes
  - Cof-constructs form basic block structure

- Effect of executing process modeled by call statement
  \[ \text{call}(p_l)(\Theta) = \Theta', \text{ with } (\Theta, \text{start}(\Pi_l), \Pi_l) \rightarrow_{\text{seq}}^* (\Theta', \zeta_{\text{sus}}, \Pi_l) \]

- Reactivation of process “controlled” by \textit{guard}

- Repeated execution controlled via disjunction of process \textit{guards}

- Mapping of model to cfg enables use of data-flow analyses on HDLs
Modeling Simulation Time

- State transitions and simulation time of utmost importance for timing analysis
- In synchronous designs, state changes scheduled on rising/falling edge of a global clock signal
  - Signals must reside stable (0 or 1) on a clock event
- Introduce special clock routine
  - Models the effect of rising and falling events on Θ
  - Self-recursion allows analyzers to separate clock cycles

▶ Explicit modeling of clock allows analysis of synchronous designs and adds support for multiple clock domains
Analysis of Open Designs

- Introduce environment routine
  - Allows modeling of transactions on input signals
- Unguarded execution within the simulation routine
- Cfg extended by attributes expressing properties of HDL constructs and the framework
  - E.g., classification of edges and assignments, prefix notation of expressions, definition/use classification
- Analyzers building on this framework are aware of concrete semantics of HDLs
Usability w.r.t. Timing Model Derivation

1. Reset analysis
   - Determines signal values at the initial state
   - Initial state apparently not visible in specification
   - Constant propagation on extended environment

2. Assumption-based model refinement

3. Static backward slicing
Assumption-based Model Refinement

Goal

- Incorporate knowledge on specific usage of processor into model
- Identify timing-dead parts and new stable signals

Data-flow analysis

- Compute safe approximation on the range of values for each identifier
- Based on interval domain

\[ \{ f \mid f: \text{identifier} \rightarrow V_{\text{Int}} \} \cup \{ \bot, \top \}, \text{ with} \]

\[ V_{\text{Int}} \equiv (\text{Value} \times \text{Value}) \cup \{ \bot, \top \} \]

- At control-flow join: form interval hull of incoming data for all identifiers
Assumption-based Model Refinement (cont.)

Transfer functions for VHDL model nodes $m \in V_{VHDL}$

- W.l.o.g, $l \vdash eval(expr(m)) = U$
- At variable assignment $v := expr$:
  
  $l[v ← U]$

- At signal assignment $s <= expr$:
  
  $l[s ← U]$

- At true/false edges $e = (m, n)$:

  $$l' = \begin{cases} 
  l & \text{if } U = T \lor U = cat(e), \\
  \bot & \text{if } U \neq cat(e), \\
  l & \text{otherwise.}
  \end{cases}$$
Assumption-based Model Refinement (cont.)

Transfer functions for framework nodes $m \in V_{\text{framework}}$

- At environment:
  $$\forall a \in \text{Assume} : l[a \leftarrow A(a), a \leftarrow A(a), \overline{a} \leftarrow A(a)]$$

- At rising edge:
  $$l[\text{clk} \leftarrow [0, 0], \text{clk} \leftarrow [1, 1], \overline{\text{clk}} \leftarrow [1, 1]]$$

- At synchronize:
  $$\forall s \in \text{Signals} : l[s \leftarrow l(s), s \leftarrow l(\overline{s})]$$

- At true/false edges $e = (m, n)$ of process guard:
  $$l' = \begin{cases} 
  l & \text{if } l \vdash \text{guard}(m) = \top, \\
  \perp & \text{if } l \vdash \text{guard}(m) \neq \text{cat}(e), \\
  l & \text{otherwise.}
  \end{cases}$$
Assumption-based Model Refinement (cont.)

Identification of stable identifiers

- Signals that get assigned the same single value under rising and falling clock events are stable

Identification of timing-dead parts

- Mark assignments to stable identifiers as timing dead
- Use restricted co-domain knowledge to identify dead outcomes of conditionals
- Mark conditional also as timing dead, if outcome is known statically
Static Backward Slicing

Goal

- Aid user in hardware model understanding
- Support timing-dead code elimination

Slicing

- Based on slicing criterion $C = (n, U)$, with $n \in V$ and $U \subseteq \text{def}(n) \cup \text{use}(n)$
- Slice with respect to $C$ is a subset $S \subseteq V_{\text{VHDL}}$ s.t. $S$ computes the same values for all $u \in U$ at $n$ as the original VHDL
- Computation of slices on VHDL requires knowledge of
  - Flow dependence between statements
  - Control dependence between statements
  - Activation dependence between processes
Static Backward Slicing (cont.)

Reconstructing flow dependencies

- Computable via *reaching definition* analysis
- For non-scalar identifiers: differ between *must* and *may* updates
  - $s \leq \ldots$ definitive assignment = must update
  - $s(1) \leq \ldots$ only partial change of composite signal = may update

Reconstructing activation dependencies

- “Special” form of flow dependence
- Use only implicitly given in form of guard statement

Reconstructing control dependencies

- Combination of *dominator* and *post-dominator* analyses
- Set of nodes, $n$ is control-dependent on, computable as

\[
\text{ctrl}(n) = \{ m | m \in MFP_{\text{dom}}(n) : n \notin MFP_{\text{pdom}}(m) \} 
\]
Static Backward Slicing (cont.)

Slicing algorithm for criterion $C = (n, U)$

\[ wset = \{(n, u) \mid u \in U\} \]
\[ vset = \emptyset \]

while ($wset \neq \emptyset$)

\[ (m, w) \leftarrow wset \]
\[ vset = vset \cup \{(m, w)\} \cup \{(c, _) \mid c \in ctrl(m)\} \]
\[ tset = \{m\} \cup ctrl(m) \cup \{act(m)\} \]
\[ wset = wset \setminus \{(m, w)\} \]
\[ \cup \bigcup_{o \in tset, u \in use(o)} \{(x, u) \mid x \in MFP_{rd}(o)(u)\} \setminus vset \]

\[ slice = \{m \mid (m, w) \in vset\} \]

- Interactive variant supports model understanding
- Slice for “instruction retires” yields all timing-relevant program points
Static Backward Slicing (cont.)

Slicing algorithm for criterion $C = (n, U)$

\[ wset = \{(n, u) \mid u \in U\} \]
\[ vset = \emptyset \]

while ($wset \neq \emptyset$)

\[
(m, w) \leftarrow wset \\
vset = vset \cup \{(m, w)\} \cup \{(c, \_ ) \mid c \in ctrl(m)\} \\
tset = \{m\} \cup ctrl(m) \cup \{act(m)\} \\
wset = wset \setminus \{(m, w)\} \\
\quad \quad \quad \cup \left( \bigcup_{o \in tset, u \in use(o)} \{(x, u) \mid x \in MFP_{rd}(o)(u)\} \setminus vset \right) \\
\]

\[ slice = \{m \mid (m, w) \in vset\} \]

- Interactive variant supports model understanding
- Slice for “instruction retires” yields all timing-relevant program points
Conclusion

- Methodology for the derivation of sound timing models

- Introduced an abstract semantics for HDLs

- Proposed a sound framework for static analysis of HDLs
  - Supports analysis of synchronous and asynchronous designs
  - Supports analysis of open and closed designs

- Enables use of program analyses on HDLs

- Building on that framework, different analyses have been presented

- Successfully used in the derivation toolset implementing the derivation methodology (cf. [Pister12])