Data Flow and Control Optimizations for Hardware and Software Co-Synthesis in Embedded Systems

Bassam Tabbara
Alberto Sangiovanni-Vincentelli
University of California at Berkeley

Embedded System

- Electronic “brain” found in many applications e.g.
  - Consumer electronics
  - Telecommunications
- Consists of:
  - Software: flexibility
  - Hardware: performance
- Application requirements on the system:
  - Small
  - Efficient
  - Power
  - Other metrics
Hardware/Software Co-design

Control dominated applications
- **Focus**: reactive controllers (e.g. car brake controller)
- **Model**: control-based (e.g. FSM)
- **Optimization**: control
- **Representative**
  - POLIS [Chiodo, 94]
  - CHINOOK [Borriello, 95]

Data dominated applications
- **Focus**: data processing (e.g. digital video TV)
- **Model**: data-based (e.g. data dependency graph)
- **Optimization**: data flow
- **Representative**
  - VULCAN [Gupta, 95]
  - COSYMA [Ernst, 93]

*Typical control-dominated applications are not purely control; they have data computations as well...*
Challenges in Control-dominated Co-Synthesis

- Experience, and feedback from POLIS users
  (automotive, telecommunications,...)
  - Cannot just focus on improved *productivity*
  - Applications with data computations result in:
    - Poor synthesized output quality: *Large* and *Inefficient*

- Co-design environments that focus on small reactive controllers suffer from this as well (TOSCA [Borriello 95], SCENIC [Gupta 97], ...)

Research Opportunity

- Incorporate data flow optimization into control dominated co-synthesis flows
  - Perform in an unbiased manner towards HW or SW
- Reduce control (e.g. y= 1; if (y==1) ... else ...)
- *Improve* the quality of the synthesized output
  - Small (code size of software, area of hardware), and
  - Efficient (performance)
  - Power, domain-specific metrics
Overview

- **Problem Statement**: Current methodologies for designing control-dominated hardware-software systems suffer from inefficient hardware and software synthesis.

- **Research Objective**: Develop a methodology that incorporates data flow in addition to control optimizations in a hardware and software co-design environment in order to improve synthesis quality.

Assumptions

- **Target**: heterogeneous control-dominated embedded system applications
  - Functional decomposition captures design as a network of Finite State Machines extended with data computations (EFSMs).

- **Focus**: representation, optimization, and synthesis of each individual task.

- No assumptions on how tasks are composed in the whole system.
Background

Reactive System Co-synthesis

CDFG is suitable for describing EFSM reactive behavior but
- Some of the control flow is hidden
- Data cannot be propagated

S1
\[ a := a + 1 \]

S0
\[ a := 5 \]

S2
\[ a := a + 1 \]

CDFG is suitable for describing EFSM reactive behavior but
- Some of the control flow is hidden
- Data cannot be propagated
Data Flow Optimization

EFSM Representation

S0
a := 5

S1
a := a + 1
a := 6

S2

Optimized EFSM Representation

Suitable Design Representation
Intermediate Design Representation

- Develop Function Flow Graph (FFG) / C-Like Intermediate Format (CLIF)
  - Able to represent EFSM
  - Suitable for data flow analysis

Function Flow Graph (FFG)

- is a triple $G = (V, E, N_0)$ where
  - $V$ is a finite set of nodes
  - $E = (x, y)$, a subset of $V \times V$, is an edge from $x$ to $y$ where $x \in Pred(y)$, the set of predecessor nodes of $y$.
  - $N_0 \in N$ is the start node corresponding to the EFSM initial state.
  - An unscheduled sequence of operations is associated with each node $N$.
  - Operations consist of TESTs performed on the EFSM inputs and internal variables, and ASSIGNs on the EFSM outputs and internal variables
FFG / CLIF Example

Legend: constant, output flow, dead operation
S# = State, S#L# = Label in State S#

C-Like Intermediate Format (CLIF)

- Import/Export Function Flow Graph (FFG)
- Unscheduled sequence of TEST and ASSIGN operations
  - \[ if \ (condition) \] goto label
  - dest = op(src)
    - op = {not, minus, ...}
  - dest = src1 op src2
    - op = {+, *, /, ||, &&, |, &, ...}
- No aliasing (no side effects)
- Loops are present
Data Flow and Control Optimization

Optimizing FFG / CLIF

Optimization Approach

- Develop *optimizer* for FFG (CLIF) intermediate design representation
- Goal: Optimize for speed, and size by reducing
  - ASSIGN operations
  - TEST operations
  - variables
- Reach goal by solving sequence of *data flow problems* for analysis and information gathering using an underlying *data flow analysis framework*
Sample Data Flow Problem
Available Expressions Example

• Goal is to eliminate re-computations

Formulate Available Expressions Problem
– Global version of common sub-expression

\[ \text{AE} = \{a + 1\} \]

AE = Available Expression

Data Flow Problem Instance

• A particular (problem) instance of a monotone data flow analysis framework is a pair \( I = (G, M) \) where \( M: N \rightarrow F \) is a function that maps each node \( N \) in \( V \) of FFG \( G \) to a function in \( F \) on the node label semilattice \( L \) of the framework \( D \).
Data Flow Analysis Framework

- A monotone data flow analysis framework $D = (L, \land, F)$ is used to manipulate the data flow information by interpreting the node labels on $N$ in $V$ of the FFG $G$ as elements of an algebraic structure where
  - $L$ is a bounded semilattice with meet $\land$, and
  - $F$ is a monotone function space associated with $L$.

Solving Data Flow Problems

- Data Flow Equations

\[
In(S3) = \bigcap_{P \in \{S1, S2\}} Out(P)
\]

\[
Out(S3) = (In(S3) - Kill(S3)) \cup Gen(S3)
\]
Solving Data Flow Problems

- Solve data flow problems using the iterative method [Kildall, 73] [Kennedy, 76]
  - General: does not depend on the flow graph
  - Optimal for a class of data flow problems [Kildall, 73]
  - Reaches fixpoint in polynomial time (O(n^2))

Data Flow Problems

- Solve following problems in order to improve design:
  - Reaching Definitions and Uses
  - Available Expression Computation
  - Copy Propagation, and constant folding
  - Reachability Analysis
- Code Improvement techniques
  - Dead Operation Elimination
  - Computation sharing through normalization
  - Code motion
Related Work

- High Level Synthesis (Silicon Compilers)
  - Local optimizations
  - Focus mainly on scheduling, allocation for hardware
    - e.g. YSC [Brayton et. al., 88]; CATHEDRAL II [Rabaey et. al., 88]

- Global Data Flow Optimization Techniques (Software Compilers)
  - Focus on handwritten code, and compilers for assembly code generation
    - e.g. [Kildall, 1973]; [Kam and Ullman, 1976]; [Aho, Sethi, and Ullman, 1988]

✓ Can be applied on the design representation and specialized to the reactive embedded domain

Specializing Problems to the Reactive Embedded Domain

```
input inp;
output outp;
int a = 0;
int _CONST_0 = 0;
int _T11 = 0;
int _T13 = 0;

S1: goto S2;
S2: a = inp;
    _T13 = a + _CONST_0;
    _T11 = a + a;
    outp = _T11;
    goto S3;
S3: outp = _T13;
    goto S3;
```
Architecture Dependent Optimizations

Architecture Independent

Architecture Dependent Optimizations

Architectural Information

lib

EFSM  FFG  OFFG  MFFG  CDFG

Sum
Sharing Sub-expressions

- Available Expressions cannot eliminate \( T_2 \).
- But if variables are registered (additional architectural information) we can share \( T_1 \) and \( T_2 \).

\[
\begin{align*}
S_1 & : T_1 = a + b; \\
& x = T_1 \\
& a = c
\end{align*}
\]

\[
\begin{align*}
S_2 & : T_2 = a + b; \\
& \text{Out} = T(a+b)
\end{align*}
\]

\[
\text{emit(Out)}
\]

Function Architecture Co-design

- System Constraints
- System Specs
- Decomposition
- Instruction Selection
- Operator Strength Reduction
- Instruction Selection
- Decomposition

\[
\begin{align*}
t_1 &= 3*b \\
t_2 &= t_1 + a \\
\text{emit} \ x(t_2)
\end{align*}
\]
Co-Synthesis Flow

- **FFG Interpreter (Simulation)**
- **EFSM** → **FFG** → **OFFG** → **CDFG SHIFT**

Or

- **Software Compilation** → **Object Code (.o)**
- **Hardware Synthesis** → **Netlist**

Design Representation in POLIS

- **SHIFT**
  - Hierarchical netlist of EFSMs
  - EFSMs represented as
    - Input/Output/State signals
    - Tabular description of transition relation
  - Stateless arithmetic, or Boolean functions
    - *Sub-circuits*

- **CDFG**
  - Each path in CDFG is an EFSM *transition*
  - *Operations*
## Results
### Synthesized Software

<table>
<thead>
<tr>
<th>Quicksort [Aho, 1988]</th>
<th>CDFG</th>
<th>Speed (68HC11 cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># nodes</td>
<td>build (sec)</td>
</tr>
<tr>
<td>EFSM $\rightarrow$ CDFG</td>
<td>336</td>
<td>2.0</td>
</tr>
<tr>
<td>EFSM $\rightarrow$ FFG $\rightarrow$ CDFG</td>
<td>206</td>
<td>0.3</td>
</tr>
<tr>
<td>Result (%)</td>
<td>38.7</td>
<td>85.0</td>
</tr>
</tbody>
</table>

## Synthesized Hardware

<table>
<thead>
<tr>
<th>SIMPLE</th>
<th>Before Hardware Optimization</th>
<th>After Hardware Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodes (BLIF)</td>
<td>literals (sop)</td>
</tr>
<tr>
<td>EFSM $\rightarrow$ CDFG</td>
<td>353</td>
<td>3204</td>
</tr>
<tr>
<td>EFSM $\rightarrow$ CLIF $\rightarrow$ CDFG</td>
<td>236</td>
<td>2415</td>
</tr>
<tr>
<td>Result (%)</td>
<td>33.1</td>
<td>24.6</td>
</tr>
</tbody>
</table>
Conclusion

- New design representation (FFG/CLIF) able to capture EFSM and permit global data flow analysis for design optimization
- Data flow and control optimization approach to hardware and software co-synthesis of embedded systems
  - Architecture independent EFSM optimizations
    - TEST, and ASSIGN operations, and variable reduction
  - Architecture dependent EFSM optimizations
    - Function/Architecture Co-design

Co-Design Methodology