HW/SW Synthesis

Outline

- Synthesis
- CFSM Optimization
- Software synthesis
  - problem
  - task synthesis
  - performance analysis
  - task scheduling
  - compilation
Aptix Board Consist of
- micro of choice
- FPGA’s
- FPIC’s

POLIS Methodology

Hardware - Software Architecture

- Hardware:
  - currently:
    - one micro-controller
    - ASICs (FPGAs)
  - in the future: several micro-controllers, DSPs, ...

- Software:
  - Set of concurrent tasks
  - Customized Real-Time Operating System

- Interfaces:
  - Hardware modules
  - Software procedures (polling, interrupt handlers, ...)
System Partitioning

POLIS Methodology
Software synthesis

- Two-level process
  - “technology” (processor) independent:
    - best decision/assignment sequence given CFSM
  - “technology” (processor) dependent:
    - conversion into machine code
      - instruction selection
      - instruction scheduling
      - register assignment
      (currently left to compiler)
    - need performance and cost analysis
      - Worst Case Execution Time
      - code and data size

Software synthesis

- Technology-independent phase:
  - construction of Control-Data Flow Graph from CFSM
    (based on BDD representation of Transition Function)
  - optimization of CDFG for
    - execution speed
    - code size
    (based on BDD sifting algorithm)
- Technology-dependent phase:
  - creation of (restricted) C code
  - cost and performance analysis
  - compilation
Software Implementation

Problem

- **Input:**  
  - set of tasks (specified by CFSMs)  
  - set of timing constraints (e.g., input event rates and response constraints)
- **Output:**  
  - set of procedures that implement the tasks  
  - scheduler that satisfies the timing constraints
- **Minimizing:**  
  - CPU cost  
  - memory size  
  - power, etc.

Software Implementation

- **How to do it ?**
- **Traditional approach:**  
  - hand-coding of procedures  
  - hand-estimation of timing input to scheduling algorithms
- **Long and error-prone**
- **Our approach: three-step automated procedure:**  
  - synthesize each task separately  
  - extract (estimated) timing  
  - schedule the tasks
- **Customized RT-OS (scheduler + drivers)**
Software Implementation

- **Current strategy:**
  - Iterate between synthesis, estimation and scheduling
  - Designer chooses the scheduling algorithm

- **Future work:**
  - Top-down propagation of timing constraints
  - Software synthesis under constraints
  - Automated scheduling selection
    (based on CPU utilization estimates)

Software synthesis procedure

- Specification, partitioning
- S-graph synthesis
- Timing estimation
- Scheduling, validation
- Code generation
- Compilation
- Testing, validation
- Production

feasible

not feasible

fail

pass
Task implementation

- Goal: quick response time, within timing and size constraints
- Problem statement:
  - Given a CFSM transition function and constraints
  - Find a procedure implementing the *transition function* while meeting the constraints
- The procedure code is acyclic:
  - powerful optimization and analysis techniques
  - looping, state storage etc. are implemented outside (in the OS)

SW Modeling Issues

- The software model should be:
  - Low-level enough to allow detailed optimization and estimation
  - High-level enough to avoid excessive details e.g. register allocation, instruction selection
- Main types of “user-mode” instructions:
  - data movement
  - ALU
  - conditional/unconditional branches
  - subroutine calls
- RTOS handles I/O, interrupts and so on
SW Modeling Issues

- Focus on control-dominated applications
  - Address only CFSM control structure optimization
  - Data path left as “don’t touch”
- Use Decision Diagrams (Bryant ‘86)
  - Appropriate for control-dominated tasks
  - Well-developed set of optimization techniques
  - Augmented with arithmetic and Boolean operators, to perform data computations

ROBDDs

- Reduced Ordered BDDs [Bryant 86]
  - A node represents a function given by the Shannon decomposition
    \[ f = x f_x + \bar{x} f_x \]
  - Variable appears once on any path from root to terminal
  - Variables are ordered
  - No two vertices represent the same function
  - Canonical
    - Two functions are equal if and only if their BDDs are isomorphic ⇒ direct application in equivalence checking
ROBDDs and Combinational Verification

- Given two circuits:
  - Build the ROBDDs of the outputs in terms of the primary inputs
  - Two circuits are equivalent if and only if the ROBDDs are isomorphic
- Complexity of verification depends on the size of ROBDDs
  - Compact in many cases

ROBDDs and Memory Explosion

- ROBDDs are not always compact
  - Size of an ROBDD can be exponential in number of variables
  - Can happen for real life circuits also
    - e.g. Multipliers

Commonly known as:
Memory Explosion Problem of ROBDDs
Technique For Handling ROBDD Memory Explosion

All the representations are canonical ⇒ combinational equivalence checking

Handling Memory Explosion: Variable Ordering

- BDD size very sensitive to variable ordering
  \[ a_1b_1 + a_2b_2 + a_3b_3 \]

Good Ordering: 8 nodes  Bad Ordering: 16 nodes
Handling Memory Explosion: Variable Ordering

- Good static as well as dynamic ordering techniques exist
- Dynamic variable reordering [Rudell 93]
  - Change variable order automatically during computations
  - Repeatedly swap a variable with adjacent variable
  - Swapping can be done locally
  - Select the best location

\[ a_1b_1 + a_2b_2 \]

SW Model: S-graphs

- Acyclic extended decision diagram computing a transition function
- S-graph structure:
  - directed acyclic graph
  - set of finite-valued variables
  - TEST nodes evaluate an expression and branch accordingly
  - ASSIGN nodes evaluate an expression and assign its result to a variable
  - Basic block + branch is a general CDFG model
    (but we constrain it to be acyclic for optimization)
**An example of S-graph**

- input event c
- output event y
- state int a
- input int b
- forever
  - if (detect(c))
    - if (a < b)
      - a := a + 1
      - emit(y)
    - else
      - a := 0
      - emit(y)

**S-graphs and functions**

- Execution of an s-graph computes a function from a set of input and state variables to a set of output and state variables:
  - Output variables are initially undefined
  - Traverse the s-graph from BEGIN to END

- Well-formed s-graph:
  - every time a function depending on a variable is evaluated, that variable has a defined value

- How do we derive an s-graph implementing a given function?
### S-graphs and functions

- **Problem statement:**
  - Given: a finite-valued multi-output function over a set of finite-valued variables
  - Find: an s-graph implementing it
- **Procedure based on Shannon expansion**
  \[ f = x f_x + x' f_x', \]
- **Result heavily depends on ordering of variables in expansion**
  - inputs before outputs: TESTs dominate over ASSIGNs
  - outputs before inputs: ASSIGNs dominate over TESTs

### Example of S-graph construction

\[ x = a \ b + c \]
\[ y = a \ b + d \]

Order: a, b, c, d, x, y
(inputs before outputs)
Example of S-graph construction

\[
x = a \cdot b + c \\
y = a \cdot b + d
\]

Order: \( a, b, x, y, c, d \) 
(interleaving inputs and outputs)

S-graph optimization

- General trade-off:
  - TEST-based is faster than ASSIGN-based (each variable is visited at most once)
  - ASSIGN-based is smaller than TEST-based (there is more potential for sharing)

- Implemented as constrained sifting of the Transition Function BDD

- The procedure can be iterated over s-graph fragments:
  - local optimization, depending on fragment criticality (speed versus size)
  - constraint-driven optimization (still to be explored)
From S-graphs to instructions

- TEST nodes → conditional branches
- ASSIGN nodes → ALU ops and data moves
- No loops in a single CFSM transition
  - (user loops handled at the RTOS level)
- Data flow handling:
  - “don’t touch” them (except common subexpression extraction)
  - map expression DAGs to C expressions
  - C compiler allocates registers and select opcodes
- Need source-level debugging environment (with any of the chosen entry languages)

Software synthesis procedure

1. Specification, partitioning
2. S-graph synthesis
3. Timing estimation
4. Scheduling, validation
5. Code generation
6. Compilation
7. Testing, validation
8. Production

feasible → fail → not feasible → pass → fail → not feasible → feasible → pass
**Software Estimation**

- No concept of SW
- Fast with Moderate Accuracy and Low Cost
- Accurate at any cost

**System Level Software Model**

- Must be fast - whole system simulation
- Processor model must be cheap
  - “what if” my processor did X
  - future processors not yet developed
  - evaluation of processor not currently using
- Must be convenient to use
  - no need to compile with cross-compilers
  - debug on my desktop
- Must be accurate enough for the purpose
What is software estimation for?

- Architectural evaluation
  - processor selection
  - bus capacity
- Partitioning evaluation
  - HW/SW partition
  - co-processor needs
- System metric evaluation
  - performance met?
  - power met?

Accuracy vs Performance vs Cost

<table>
<thead>
<tr>
<th></th>
<th>Accuracy</th>
<th>Speed</th>
<th>$$$*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Emulation</td>
<td>+++</td>
<td>++</td>
<td>---</td>
</tr>
<tr>
<td>Cycle accurate model</td>
<td>++</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Cycle counting ISS</td>
<td>++</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Dynamic estimation</td>
<td>+</td>
<td>+++</td>
<td>++</td>
</tr>
<tr>
<td>Static spreadsheet</td>
<td>-</td>
<td>++++</td>
<td>+++</td>
</tr>
</tbody>
</table>

*$$=$= nre + per model + per design
Software Performance Estimation
Problem (1)

• Performance analysis for general programs
  • Need to analyze the structure of given program.
  • Available for limited structures [Stoyenko86, Koza89].
  • Need user’s annotations [Park93].

Dynamic Estimation

■ DSP Processors
  • relatively data independent
  • most time spent in hand-coded kernels
  • static data-flow consumes most cycles
  • small number of threads, simple interrupts

■ Regular processors
  • arbitrary C, highly data dependent
  • commercial RTOS, many threads
  • complex interrupts, priorities
Software Performance Estimation Problem (2)

* Software estimation problems in HW/SW co-design
  • The structure and behavior of synthesized programs are known in the co-design system.
  • Quick (and as accurate as possible) estimation methods are needed.
    • Quick methods for HW/SW partitioning [Hu94, Gupta94].
    • Accurate method using a timing accurate co-simulation [Henkel93].

Conventional System Design Flow

- system partition
  - design criteria:
    - performance
    - cost
    - modifiability
    - testability
    - reliability
- HW design
- SW design
- requirements
- re-partitioning
- performance tuning
- system debug
- performance analysis
- Long iteration loop!!
POLIS : Software Synthesis

POLIS : S-graph Level Estimation
Problems in Software Performance Estimation

How to link behavior to assembly code?
- Model C code generated from S-graph
  and use a set of cost parameters

How to handle the variety of compilers and CPUs?

Software Model
Execution time of a path and the code size

**Property**: Form of each statement is determined by type of corresponding node.

\[
T_{\text{struct}} = f^c \pi_i C_t(\text{node type of } (i), \text{variable type of } (i))
\]

- \( \pi_i \): takes value 1 if node \( i \) is on a path, otherwise 0.

- \( C_t(n,v) \): execution time for node type \( n \) and variable type \( v \).

\[
S_{\text{struct}} = f^c C_s(\text{node type of } (i), \text{variable type of } (i))
\]

- \( C_s(n,v) \): code size for node type \( n \) and variable type \( v \).

Cost Parameters

* Pre-calculated cost parameters for:

1. \( C_t(n,v), C_s(n,v) \):
   Execution time and code size for node type \( n \) and variable type \( v \).

2. \( T_{pp}, S_{pp} \):
   Pre- and post- execution time and code size.

3. \( T_{\text{init}}, S_{\text{init}} \):
   Execution time and code size for local variable initialization.
Problems in Software Performance Estimation

How to link behavior to assembly code?

How to handle the variety of compilers and CPUs?
- \( \rightarrow \) prepare cost parameters for each target

Extraction of Cost Parameters

set of benchmark programs

<table>
<thead>
<tr>
<th>target C compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>static analyzer</td>
</tr>
<tr>
<td>execution &amp; profiling</td>
</tr>
</tbody>
</table>

parameter extractor

| cost parameters |
Algorithm

- Preprocess: extracting set of cost parameters.
- Weighting nodes and edges in given S-graph with cost parameters.
- Traversing weighted S-graph.
- Finding maximum cost path and minimum cost path using Depth-First Search on S-graph.
- Accumulating 'size’ costs on all nodes.

S-graph Level Estimation :Algorithm

Cost C is a triple (min_time, max_time, code_size)

Algorithm: SGtrace (sg)

\[
\text{if (sgi == NULL) return (C(0, ,0));}
\]

\[
\text{if (sgi has been visited)}
\]

\[
\text{return ( pre-calculated C(*,*),0) associated with sgi );}
\]

\[
\text{C_i = initialize (max_time = 0, min_time = , code_size = 0);}
\]

\[
\text{for each child sg_j of sg_i {}
\]

\[
\text{C_{ij} = SGtrace (sg_j) + edge cost for edge e_{ij};}
\]

\[
\text{C_i.max_time = max(C_i.max_time, C_{ij}.max_time);}
\]

\[
\text{C_i.min_time = min(C_i.min_time, C_{ij}.min_time);}
\]

\[
\text{C_i.code_size += C_{ij}.code_size;}
\]

\}

\[
\text{C_i += node cost for node sgi;}
\]

\[
\text{return (C_i);}
\]
Experiments

* Proposed methods implemented and examined in POLIS system.

* Target CPU and compiler: M68HC11 and Introl C compiler.

* Difference D is defined as

\[ D = \frac{\text{cost estimated} - \text{cost measured}}{\text{cost measured}} \]

Experimental Results : S-graph Level
Performance and cost estimation: Summary

- S-graph: low-level enough to allow accurate performance estimation
- Cost parameters assigned to each node, depending on:
  - system type (CPU, memory, bus, ...)
  - node and expression type
- Cost parameters evaluated via simple benchmarks
  - need timing and size measurements for each target system
  - currently implemented for MIPS, 68332 and 68HC11 processors

Performance and cost estimation

- Example: 68HC11 timing estimation
- Cost assigned to s-graph edges
  - (different for taken/not taken branches)
- Estimated time:
  - min: 26 cycles
  - max: 126 cycles
- Accuracy: within 20% of profiling
Open Problems

- Better synthesis techniques
  - add state variables to simplify s-graph
  - performance-driven synthesis of critical paths
  - exact memory/speed trade-off
- Estimation of caching and pipelining effects
  - may have little impact on control-dominated systems
    (frequent branches and context switches)
  - relatively easy during co-simulation

Software synthesis procedure

1. Specification, partitioning
2. S-graph synthesis
3. Timing estimation
4. Scheduling, validation
5. Code generation
6. Compilation
7. Testing, validation
8. Production