Software Estimation and Synthesis

Thanks to Prof. Sharad Malik at Princeton University and Prof. Reinhard Wilhelm at Universitat des Saarlandes for some of the slides
Outline

• SW estimation overview
• Program path analysis
• Micro-architecture modeling
• Implementation examples: Cinderella
• SW estimation in VCC
• SW estimation in AI
• SW estimation in POLIS
SW estimation in HW-SW co-design

- **Architecture Function Mapping**
  - No concept of SW
  - Fast with Moderate Accuracy and Low Cost

- **Capacity**
  - Accurate at any cost
SW estimation overview: motivation

• SW estimation helps to
  – Evaluate HW/SW trade-offs
  – Check performance/constraints
    – Higher reliability
  – Reduce system cost
    – Allow slower hardware, smaller size, lower power consumption
SW estimation overview

• SW 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]
SW estimation overview: tasks

• Architectural evaluation
  – processor selection
  – bus capacity

• Partitioning evaluation
  – HW/SW partition
  – co-processor needs

• System metric evaluation
  – performance met?
  – power met?
  – size met?
SW estimation overview: Static vs. Dynamic

• Static estimation
  – Determination of runtime properties at compile time
  – Most of the (interesting) properties are undecidable => use approximations
  – An approximation program analysis is safe, if its results can always be depended on. Results are allowed to be imprecise as long as they are not on the safe side
  – Quality of the results (precision) should be as good as possible
SW estimation overview: Static v.s. Dynamic

• Dynamic estimation
  – Determination of properties at runtime
  – 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
SW estimation overview: approaches

• Two aspects to be considered
  – The structure of the code (program path analysis)
    – E.g. loops and false paths
  – The system on which the software will run (micro-architecture modeling)
    – CPU (ISA, interrupts, etc.), HW (cache, etc.), OS, Compiler

• Needs to be done at high/system level
  – Low-level
    – e.g. gate-level, assembly-language level
    – Easy and accurate, but long design iteration time
  – High/system-level
    – Reduces the exploration time of the design space
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 used
- Must be convenient to use
  - no need to compile with cross-compilers
  - debug on my desktop
- Must be accurate enough for the purpose
## 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*
Outline

- SW estimation overview
- Program path analysis
- Micro-architecture modeling
- Implementation examples: Cinderella
- SW estimation in VCC
- SW estimation in AI
- SW estimation in POLIS
Program path analysis

- **Basic blocks**
  - A basic block is a program segment which is only entered at the first statement and only left at the last statement.
  - Example: function calls
  - The WCET (or BCET) of a basic block is determined

- **A program is divided into basic blocks**
  - Program structure is represented on a directed program flow graph with basic blocks as nodes.
  - A longest / shortest path analysis on the program flow identify WCET / BCET
Program path analysis

- Program path analysis
  - Determine extreme case execution paths.
  - Avoid exhaustive search of program paths.

```c
for (i=0; i<100; i++) {
    if (rand() > 0.5)
        j++;
    else
        k++;
}
```

- Eliminate False Paths:
  - Make use of path information provided by the user.

```c
if (ok)
    i = i*i + 1;
else
    i = 0;
if (i)
    j++;
else
    j = j*j;
```

$2^{100}$ possible worst case paths!

Always executed together!
Program path analysis

• Path profile algorithm
  – Goal: Determines how many times each acyclic path in a routine executes
  – Method: identify sets of potential paths with states
  – Algorithms:
    – Number final states from 0, 1, to \( n-1 \), where \( n \) is the number of potential paths in a routine; a final state represents the single path taken through a routine
    – Place instrumentation so that transitions need not occur at every conditional branch
    – Assign states so that transitions can be computed by a simple arithmetic operation
    – Transforms a control-flow graph containing loops or huge numbers of potential paths into an acyclic graph with a limited number of paths
Program path analysis

- Transform the problem into an integer linear programming (ILP) problem.
  - Basic idea:

\[
\max \left( \sum_{i} c_i x_i \right)
\]

subject to a set of linear constraints that bound all feasible values of \( x_i \)’s.

Assumption for now: simple micro-architecture model

(constant instruction execution time)
Program path analysis: structural constraints

- Linear constraints constructed automatically from program’s control flow graph.

Example: While loop

```c
/* p >= 0 */
q = p;
while (q<10)
  q++;
  r = q;
```

Structural Constraints
At each node:

- Exec. count of \( B_i \) = \( \sum \text{inputs} = \sum \text{outputs} \)

```
[1] \( x_1 \)
[2] \( x_2 = d_2 + d_4 = d_3 + d_5 \)
[3] \( x_3 = d_3 = d_4 \)
[4] \( x_4 = d_5 = d_6 \)
[5] \( x_3 \)
```

Functional Constraints:
- provide loop bounds and other path information

```
0 x_1 \leq x_3 \leq 10 x_1
```
Program path analysis: functional constraints

- Provide loop bounds (mandatory).
- Supply additional path information (optional).

Nested loop:
\[
\begin{align*}
  x_1 &: \text{for } (i=0; i<10; ++i) \\
  x_2 &: \text{for } (j=0; j<i; ++j) \\
  x_3 &: A[i] += B[i][j];
\end{align*}
\]

If statements:
\[
\begin{align*}
  x_1 &: \text{if } (\text{ok}) \\
  x_2 &: i=i*i+1; \quad \text{else} \\
  x_3 &: i=0; \\
  x_4 &: \text{if } (i) \\
  x_5 &: j=0; \quad \text{else} \\
  x_6 &: j=j*j;
\end{align*}
\]

\[
\begin{align*}
  x_2 &= 10x_1 & \text{loop bounds} \\
  0x_2 &\leq x_3 \leq 9x_2 & \text{path info.} \\
  x_3 &= 45x_1 \\
  x_2 &\leq 0.5x_1 \\
  B_2 \text{ and } B_5 \text{ have same execution counts:} \\
  x_2 &= x_5
\end{align*}
\]
Outline

• SW estimation overview
• Program path analysis
• Micro-architecture modeling
• Implementation examples: Cinderella
• SW estimation in VCC
• SW estimation in AI
• SW estimation in POLIS
Micro-architecture modeling

- Micro-architecture modeling
  - Model hardware and determine the execution time of sequences of instructions.
  - Caches, CPU pipelines, etc. make WCET computation difficult since they make it history-sensitive
  - Program path analysis and micro-architecture modeling are inter-related.
Micro-architecture modeling

• Pipeline analysis
  – Determine each instruction’s worst case effective execution time by looking at its surrounding instructions within the same basic block.
  – Assume constant pipeline execution time for each basic block.

• Cache analysis
  – Dominant factor.
  – Global analysis is required.
  – Must be done simultaneously with path analysis.
Micro-architecture modeling

• Other architecture feature analysis
  – Data dependent instruction execution times
    – Typical for CISC architectures
      – e.g. shift-and-add instructions
  – Superscalar architectures
Micro-architecture modeling: pipeline features

• Pipelines are hard to predict
  – Stalls depend on execution history and cache contents
  – Execution times depend on execution history

• Worst case assumptions
  – Instruction execution cannot be overlapped
  – If a hazard cannot be safely excluded, it must be assumed to happen
  – For some architectures, hazard and non-hazard must be considered (interferences with instruction fetching and caches)

• Branch prediction
  – Predict which branch to fetch based on
    – Target address (backward branches in loops)
    – History of that jump (branch history table)
    – Instruction encoding (static branch prediction)
Micro-architecture modeling: pipeline features

• On average, branch prediction works well
  – Branch history correctly predicts most branches
  – Very low delays due to jump instructions

• Branching is hard to predict
  – Depends on execution history (branch history table)
  – Depends on pipeline: when does fetching occur?
  – Incorporates additional instruction fetches not along the execution path of the program (mis-predictions)
  – Changes instruction cache quite significantly

• Worst case assumptions
  – Instruction fetches occur along all possible execution paths
  – Prediction is wrong: re-fetch along other path
  – I-Cache contents are ruined
Micro-architecture modeling: pipeline analysis

- Goal: calculate all possible pipeline states at a program point
- Method: perform a cycle-wise evolution of the pipeline, determining all possible successor pipeline states
- Implemented from a formal model of the pipeline, its stages and communication between them
- Generated from a PAG specification
- Results in WCET for basic blocks
- Abstract state is a set of concrete pipeline states; try to obtain a superset of the collecting semantics
- Sets are small as pipeline is not too history-sensitive
- Joins in CFG are set union
Micro-architecture modeling: I-cache analysis

• Extend previous ILP formulation

• Without cache analysis
  – For each instruction, determine:
    – total execution count
    – execution time
  – Instructions within a basic block have same execution counts
    – Group them together.

• With i-cache analysis
  – For each instruction, determine:
    – cache hit execution count
    – cache miss execution count
    – cache hit execution time
    – cache miss execution time
  – Instructions within a basic block may have different cache hit/miss counts
    – Need other grouping method.
Grouping Instructions: Line-blocks

- Line-block (l-block) = Basic block $\cap$ Cache set
  - All instructions within a l-block have same cache hit/miss counts.
- Construction of l-blocks:

  - Conflicting l-blocks
  - Non-conflicting l-blocks
Modified ILP Formulation

Let:

- cache hit count of l-block $B_{i,j}$
  
- cache miss count of l-block $B_{i,j}$
  
- exec. time of l-block $B_{i,j}$ given that it is a cache hit
  
- exec. time of l-block $B_{i,j}$ given that it is a cache miss
  
- number of l-blocks of basic block $B_i$

Maximize:

$$\sum_i \sum_j \left( c_{i,j} x_{i,j}^{hit} + c_{i,j} x_{i,j}^{miss} \right)$$

subject to:

$$x_i = x_{i,j}^{hit} + x_{i,j}^{miss} \quad j = 1, 2, \ldots, n_i$$

structural constraints

functionality constraints

cache constraints
Micro-architecture modeling: I-cache analysis

• Cache constraints
  – Describe cache hit/miss activities of l-blocks.
  – For each cache set:
    – if there is only one l-block $B_{i,j}$ mapping to it, then there will be at most one cache miss:
      \[ X_{i,j}^{\text{miss}} \leq 1 \]
    – if there are two or more conflicting l-blocks mapping to it, 
      *Cache Conflict Graph* is needed...
Micro-architecture modeling: I-cache analysis

Capture control flow of I-blocks mapping to the same cache set only.

Control Flow Graph (CFG)  Cache Conflict Graph (CCG)
Micro-architecture modeling: I-cache analysis

- Direct-mapped I-cache analysis

Flow at node $B_{k,l}$:

$$x_k = p(s,k,I) + p(m,n,k,I) + p(k,l,k,I)$$

$$= p(k,l,e) + p(k,l,m,n) + p(k,l,k,l)$$

Cache hit count for $l$-block $B_{k,l}$:

$$p(k,l,k,l) \leq x_{k,l}^{hit} \leq p(s,k,l) + p(k,l,k,l)$$

Starting Condition:

$$p(s,k,l) + p(s,m,n) + p(s,e) = 1$$
Micro-architecture modeling: D-cache analysis

• Difficulties:
  – Data flow analysis is required.
  – Load/store address may be ambiguous.
  – Load/store address may change.

• Simple solution:
  – Extend cost function to include data cache hit/miss penalties.
  – Simulate a block of code with known execution path to obtain data hits and misses.

```c
x_1 if (something) {
  x_2 for (i=0; i<10; ++i)
  x_3 for (j=0; j<i; ++j)
  x_4 A[i] += B[i][j];
}
else {
  x_5 /* ... */
}
```

Data hits/misses of this loop nest can be simulated.
Outline

• SW estimation overview
• Program path analysis
• Micro-architecture modeling
• Implementation examples: Cinderella
• SW estimation in VCC
• SW estimation in AI
• SW estimation in POLIS
Implementation examples: Cinderella

- Implemented platforms:
  - Intel i960KB
  - Motorola M68000

- Re-targetable back-ends:
  - Object File Module
    - Read program’s executable file.
    - Provide mapping information.
  - Instruction Set Module
    - Architecture dependent.
    - Decode binary code.
  - Machine Module
    - Micro-architecture dependent.
    - Model hardware timing properties.

- 30k lines of C++ code.

http://www.ee.princeton.edu/~yauli/cinderella-3.0
Implementation examples: Cinderella

• Timing analysis is done at machine code level.
  – read executable file directly.

• Annotation is done at source level.
  – need to map machine code to source code.
  – use debugging information stored in executable file.
  – Advantages:
    – source level, compiler independent.
    – optimum code quality.

• Reduce retargeting efforts.
  – Identify target dependent modules.
### Implementation examples: Cinderella

- Practical validation: set of programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Lines</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>Circle drawing routine</td>
<td>100</td>
<td>1,588</td>
</tr>
<tr>
<td>des</td>
<td>Data Encryption Standard</td>
<td>192</td>
<td>1,852</td>
</tr>
<tr>
<td>dhry</td>
<td>Dhrystone benchmark</td>
<td>761</td>
<td>1,360</td>
</tr>
<tr>
<td>fdct</td>
<td>Forward discrete cosine transform</td>
<td>300</td>
<td>996</td>
</tr>
<tr>
<td>fft</td>
<td>1024-point fast Fourier transform</td>
<td>57</td>
<td>500</td>
</tr>
<tr>
<td>line</td>
<td>Line drawing routine</td>
<td>165</td>
<td>1,556</td>
</tr>
<tr>
<td>sort</td>
<td>Insertion sort of 500 elements</td>
<td>41</td>
<td>152</td>
</tr>
<tr>
<td>sort2</td>
<td>Same as sort, but with inlined functions</td>
<td>30</td>
<td>148</td>
</tr>
<tr>
<td>stats</td>
<td>Calculate sum, mean and variance of two 1,000-element arrays</td>
<td>100</td>
<td>656</td>
</tr>
<tr>
<td>stats2</td>
<td>Same as stats, but with inlined functions</td>
<td>90</td>
<td>596</td>
</tr>
<tr>
<td>whetstone</td>
<td>Whetstone benchmark</td>
<td>196</td>
<td>2,760</td>
</tr>
<tr>
<td>jpeg</td>
<td>Decompression of 128x96 JPEG color image</td>
<td>857</td>
<td>5,408</td>
</tr>
</tbody>
</table>
Experimental Results

Intel i960KB Measurements

- Est. WCET w/o cache analysis
- Est. WCET with Cache Analysis
- Est. WCET with cache analysis (loop bounds only)

Program:
- circle
- dhry
- fft
- sort
- stats
- whetstone

Estimate WCET with cache analysis (loop bounds only):
- 52,515

Estimate WCET w/o cache analysis:
- 52,515
## Implementation examples: Cinderella

- ILP performance issues

<table>
<thead>
<tr>
<th>Program</th>
<th>No. of variables</th>
<th>No. of constraints</th>
<th>CPU time</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$d's$</td>
<td>$f's$</td>
<td>$p's$</td>
<td>$x's$</td>
</tr>
<tr>
<td>circle</td>
<td>8</td>
<td>1</td>
<td>81</td>
<td>100</td>
</tr>
<tr>
<td>des</td>
<td>174</td>
<td>11</td>
<td>723</td>
<td>560</td>
</tr>
<tr>
<td>dhry</td>
<td>102</td>
<td>21</td>
<td>490</td>
<td>504</td>
</tr>
<tr>
<td>fdct</td>
<td>8</td>
<td>0</td>
<td>18</td>
<td>34</td>
</tr>
<tr>
<td>fft</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>80</td>
</tr>
<tr>
<td>line</td>
<td>31</td>
<td>2</td>
<td>264</td>
<td>231</td>
</tr>
<tr>
<td>sort</td>
<td>15</td>
<td>1</td>
<td>0</td>
<td>58</td>
</tr>
<tr>
<td>sort2</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>stats</td>
<td>28</td>
<td>13</td>
<td>57</td>
<td>180</td>
</tr>
<tr>
<td>stats2</td>
<td>28</td>
<td>7</td>
<td>41</td>
<td>144</td>
</tr>
<tr>
<td>whetstone</td>
<td>52</td>
<td>3</td>
<td>258</td>
<td>388</td>
</tr>
<tr>
<td>jpeg</td>
<td>296</td>
<td>20</td>
<td>1,816</td>
<td>1348</td>
</tr>
</tbody>
</table>

CPU Times are measured on a SGI Indigo2 workstation.
Outline

- SW estimation overview
- Program path analysis
- Micro-architecture modeling
- Implementation examples: Cinderella
- SW estimation in VCC
- SW estimation in AI
- SW estimation in POLIS
SW estimation in VCC

Objectives

• To be faster than co-simulation of the target processor (at least one order of magnitude)
• To provide more flexible and easier to use bottleneck analysis than emulation (e.g., who is causing the high cache miss rate?)
• To support fast design exploration (what-if analysis) after changes in the functionality and in the architecture
• To support derivative design
• To support well-designed legacy code (clear separation between application layer and API SW platform layer)
SW estimation in VCC

Approaches

• Various trade-offs between simplicity, compilation/simulation speed and precision

• Virtual Processor Model: it compiles C source to simplified “object code” used to back-annotate C source with execution cycle counts and memory accesses
  – Typically ISS uses object code, Cadence CC-ISS uses assembly code, commercial CC-ISS’s use object code

• CABA: C-Source Back Annotation and model calibration via Target Machine Instruction Set

• Instruction-Set Simulator: it uses target object code to:
  – either reconstruct annotated C source (Compiled-Code ISS)
  – or executed on an interpreted ISS
**SW estimation in VCC**

**Scenarios**

- **Target Processor**
  - Virtual Compiler
  - Target Processor
  - Host Compiler

- **Compiled Code**
  - Virtual Instruction Set Simulator
  - Compiled Code
  - Virtual Instruction Set

- **Instruction Set**
  - VCC Virtual Processor Instruction Set
  - Target Processor Instruction Set

- **Co-simulation**
  - `tmp = b + c`
  - `c = f(d)`
  - `MT update ∆1`
  - `y = a * c + b`
  - `MT update ∆2 + ∆3`
  - `write B y`
  - `f6(y)`
  - `MT update ∆4`
  - `return ∆4`

- **ASM to C**
  - `r = (s << *a)`
  - `a = r + m * x`

**Table 3-19 Instruction Set Summary (Continued)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands Syntax</th>
<th>Condition Register CR with Complement</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ret</code></td>
<td><code>ret</code></td>
<td>Condition Register XOR</td>
<td></td>
</tr>
<tr>
<td><code>cbr</code></td>
<td><code>cbr</code></td>
<td>Condition Register CR</td>
<td></td>
</tr>
<tr>
<td><code>cbr</code></td>
<td><code>cbr</code></td>
<td>Condition Register CR</td>
<td></td>
</tr>
<tr>
<td><code>cbr</code></td>
<td><code>cbr</code></td>
<td>Condition Register CR</td>
<td></td>
</tr>
</tbody>
</table>

**Annotations**
- White Box C
- Annotated White Box C
- `Annotated White Box C.obj`
- `White Box C.obj`
- `tmp = b + c`
SW estimation in VCC

Limitations

• C (or assembler) library routine estimation (e.g. trigonometric functions): the delay should be part of the library model

• Import of arbitrary (especially processor or RTOS-dependent) legacy code
  – Code must adhere to the simulator interface including embedded system calls (RTOS): the conversion is not the aim of software estimation
SW estimation in VCC

Virtual Processor Model (VPM) compiled code virtual instruction set simulator

• Pros:
  – does not require target software development chain
  – fast simulation model generation and execution
  – simple and cheap generation of a new processor model
  – Needed when target processor and compiler not available

• Cons:
  – hard to model target compiler optimizations (requires “best in class” Virtual Compiler that can also as C-to-C optimization for the target compiler)
  – low precision, especially for data memory accesses
SW estimation in VCC

Interpreted instruction set simulator (I-ISS)

• Pros:
  – generally available from processor IP provider
  – often integrates fast cache model
  – considers target compiler optimizations and real data and code addresses

• Cons:
  – requires target software development chain
  – often low speed
  – different integration problem for every vendor (and often for every CPU)
  – may be difficult to support communication models that require waiting to complete an I/O or synchronization operation
SW estimation in VCC

Compiled code instruction set simulator (CC-ISS)

• Pros:
  – very fast (almost same speed as VPM, if low precision is required)
  – considers target compiler optimizations and real data and code addresses

• Cons:
  – often not available from CPU vendor, expensive to create
  – requires target software development chain
SW estimation in VCC

Suggested approaches

- VPM-C and CABA
  - either using a state-of-the-art compiler with a object code model
  - or using back-annotation from target object code for addresses and delays
  - suitable for: system architect, early evaluation, algorithmic optimization, coarse-grained trade-offs, resilient input stimuli behavior (data flow oriented, no target compiler available – VPM-C)

- ISS
  - CC-ISS when available
  - I-ISS otherwise
  - suitable for: detailed software design, driver design, HW/SW co-verification and integration
SW estimation in VCC

**SPA tools**

- **SPA-C**: SPA Calibration and Profiling
  - View and analysis of a C code with simulation capability of the annotated model
    - SW Developer can use it to modify algorithm code and optimize speed
    - CPU Modeler/Provider can use it to modify the cross reference file
- **SPA-V**: SPA Viewer
  - SPA-C export each simulation into an XML model. SPA-V allows to view and compare different simulations of one code.
- **VPM-C**: VPM Calibration
  - VPM-C generate the CPU model for each user C code.
SW estimation in VCC

SPA tool chain
SW estimation in VCC

VPM-C: goals

• Capitalize on top of VPM and provides an automated way to produce bss files.
• Test and validate the VPM technique across several environments and processor families.
• Measure the accuracy and the resilience of the VPM software estimation technique.
• Identify usability of VPM in a real design.
• Produce a methodology on the different steps necessary to use this VPM derived technique.
SW estimation in VCC

VPM-C: short description

• Set of tools to automate generation of calibrated CPU models for each behavioral block using the VPM technique:
  – Each behavioral block is mapped to its own CPU/DSP calibrated model.
  – Caches effects and external memories access penalties not integrated. (done using cache, bus and memory models)

• Provides 100% accuracy on the set of stimuli used during generation.

• Resilience to data variation as good as VPM Data-book technique.

• Validated on RISC, DSP, and VLIW with 40+ EEMBC test codes.

• VPM Calibrated Technique requires a target CPU Development Environment (Compiler, ISS execution/profiling, …)
SW estimation in VCC

VPM-C: processor model creation

- Initial Model (Created from Cross-Ref Only!)
- VCC Environment
- CPU Development Environment (Compiler, ISS)
- VPM Java Framework (VJF)
- CPU Specific
- Calibrated Model
- C Source and Test Vector
SW estimation in VCC

VPM-C: calibration process

Input Test Vectors obtained during a simulation

CPU model used by the mapping link

First Simulation → C-Macros are used to capture test vectors and dump them into files.
SW estimation in VCC

VPM-C: calibration process

**Architectural Diagram**

**Behavioral Diagram**

- RTOS
- CPU
- MEM

- Cpu.bss for [block]

- CPU/DSP model (i.e., cpu.bss) generated by VPM Calibration tool during the Data Learning phase

**ISS Simulator**

- IssCycles

- VccCycles

- VccError
SW estimation in VCC

VPM-C: some results

- One TriMedia model done for each individual MPEG processes calibrated on one sub stream (12 images).

<table>
<thead>
<tr>
<th>Code Name</th>
<th>ISS Cycles *</th>
<th>VCC Cycles * Before Correction</th>
<th>VCC Error Before Correction</th>
<th>VCC Code Coverage</th>
<th>VCC Cycles After Correction</th>
<th>New VCC Error **</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code 1</td>
<td>243109</td>
<td>188151</td>
<td>-22.60</td>
<td>59.38 %</td>
<td>243109</td>
<td>0</td>
</tr>
<tr>
<td>Code 2</td>
<td>23279429</td>
<td>20261361</td>
<td>+12.96</td>
<td>68.08 %</td>
<td>23279429</td>
<td>0</td>
</tr>
<tr>
<td>Code 3</td>
<td>6583</td>
<td>4585</td>
<td>+30.35</td>
<td>52.83 %</td>
<td>6583</td>
<td>0</td>
</tr>
<tr>
<td>Code 4</td>
<td>216628856</td>
<td>217218320</td>
<td>+0.27</td>
<td>78.44 %</td>
<td>216628856</td>
<td>0</td>
</tr>
<tr>
<td>Code 5</td>
<td>324811247</td>
<td>240334881</td>
<td>-26.00</td>
<td>62.63 %</td>
<td>324811247</td>
<td>0</td>
</tr>
<tr>
<td>Code 6</td>
<td>1446</td>
<td>973</td>
<td>-32.67</td>
<td>69.56 %</td>
<td>1446</td>
<td>0</td>
</tr>
<tr>
<td>Code 7</td>
<td>71906768</td>
<td>81349419</td>
<td>+13.13</td>
<td>61.36 %</td>
<td>71906768</td>
<td>0</td>
</tr>
<tr>
<td>Code 8</td>
<td>63923750</td>
<td>29367289</td>
<td>-54.05</td>
<td>81.66 %</td>
<td>63923750</td>
<td>0</td>
</tr>
<tr>
<td>Code 9</td>
<td>125314925</td>
<td>96598838</td>
<td>-22.91</td>
<td>49.00 %</td>
<td>125314925</td>
<td>0</td>
</tr>
<tr>
<td>Code 10</td>
<td>336000031</td>
<td>157095149</td>
<td>-53.24</td>
<td>71.72 %</td>
<td>336000031</td>
<td>0</td>
</tr>
<tr>
<td>Code 11</td>
<td>362040957</td>
<td>518610854</td>
<td>+43.24</td>
<td>29.26 %</td>
<td>362040957</td>
<td>0</td>
</tr>
<tr>
<td>Code 12</td>
<td>253535440</td>
<td>242771676</td>
<td>-4.24</td>
<td>78.36 %</td>
<td>253535440</td>
<td>0</td>
</tr>
</tbody>
</table>

* Total number of cycles for executing the process on reference stream
** By construction
SW estimation in VCC

CABA - VI

For each processor:

• Group target instructions into m Virtual Instructions (e.g., ALU, load, store, …)

• For each one of n (much larger than m) benchmarks
  – Run ISS and get benchmark cycle count and VIs execution count

• Derive average execution time for each VI (processor BSS file) by best fit on benchmark run data

• For each functional block:
  – Compile source and extract VI composition for each ASM Basic Block
  – Split source into BBs and back-annotate estimated execution time using ASM BBs’ VI composition and BSS
  – Run VCC and get functional block cycle count
SW estimation in VCC

CABA - VI

- CABA-VI: uses a calibration-like procedure to obtain average execution timing for each target instruction (or instruction class – Virtual Instruction (VI)). Unlike the similar VPM technique, the VI’s are target-dependent. The resulted BSS is used to generate the performance annotations (delay, power, bus traffic) and its accuracy is not limited to the calibration codes.

- In both cases, part of the CCISS infrastructure is re-used to:
  - parse the assembler,
  - identify the basic blocks,
  - identify and remove the cross-reference tags,
  - handle embedded waits and other constructs,
  - generate code for bus traffic.
SW estimation in VCC

CABA - VI

Each benchmark used for calibration generates an equation of the form:

\[ n_1 \cdot v_1 + n_2 \cdot v_2 + \cdots + n_n \cdot v_n = b \]

- \( n_i \) the number of times the instruction (or class of instructions) \( v_i \) is executed during the benchmark run
- \( v_i \) the execution time for the \( i \)th instruction
- \( b \) the total time (number of cycles) required for the execution of the benchmark
- \( n \) the number of instructions (or instruction classes)

The calibration factor is obtained from an error function:

\[ e = \frac{1}{m} \sum_{i=1}^{m} \left( \frac{B_i}{b_i} - 1 \right)^2 \]

Error Function to Minimize

\[ \begin{cases} 
  n_{11} \cdot v_1 + n_{12} \cdot v_2 + \cdots + n_{1n} \cdot v_n = b_1 \\
  n_{21} \cdot v_1 + n_{22} \cdot v_2 + \cdots + n_{2n} \cdot v_n = b_2 \\
  \vdots \\
  n_{m1} \cdot v_1 + n_{m2} \cdot v_2 + \cdots + n_{mn} \cdot v_n = b_m 
\end{cases} \]

\( v_1 > 0 \)
\( v_2 > 0 \)
\( \vdots \)
\( v_n > 0 \)
SW estimation in VCC

**Results**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Simulation</th>
<th>PSIM</th>
<th>RelErr</th>
</tr>
</thead>
<tbody>
<tr>
<td>bs_cfg</td>
<td>48053.9</td>
<td>48236</td>
<td>-0.12%</td>
</tr>
<tr>
<td>crc_cfg</td>
<td>330345</td>
<td>320862</td>
<td>2.99%</td>
</tr>
<tr>
<td>insertsort_cfg</td>
<td>480090</td>
<td>480381</td>
<td>-0.03%</td>
</tr>
<tr>
<td>jfdctint_cfg</td>
<td>1.20559e+06</td>
<td>1205844</td>
<td>-0.01%</td>
</tr>
<tr>
<td>lms_cfg</td>
<td>438952</td>
<td>430956</td>
<td>1.88%</td>
</tr>
<tr>
<td>matmul_cfg</td>
<td>1.14307e+06</td>
<td>1143308</td>
<td>-0.01%</td>
</tr>
<tr>
<td>fir_cfg</td>
<td>2.61924e+06</td>
<td>2597397</td>
<td>0.85%</td>
</tr>
<tr>
<td>fft1k_cfg</td>
<td>1.32049e+06</td>
<td>1298882</td>
<td>1.67%</td>
</tr>
<tr>
<td>fibcall_cfg</td>
<td>120073</td>
<td>120324</td>
<td>-0.10%</td>
</tr>
<tr>
<td>fibo_cfg</td>
<td>6.28005e+06</td>
<td>6280268</td>
<td>-0.00%</td>
</tr>
<tr>
<td>fftl1_cfg</td>
<td>1.00826e+06</td>
<td>984526</td>
<td>2.42%</td>
</tr>
<tr>
<td>ludcmp_cfg</td>
<td>1.9772e+06</td>
<td>1956308</td>
<td>1.07%</td>
</tr>
<tr>
<td>minver_cfg</td>
<td>1.12565e+06</td>
<td>1114693</td>
<td>0.99%</td>
</tr>
<tr>
<td>qurt_cfg</td>
<td>1.46096e+06</td>
<td>1421282</td>
<td>2.80%</td>
</tr>
<tr>
<td>select_cfg</td>
<td>824290</td>
<td>746637</td>
<td>10.42%</td>
</tr>
</tbody>
</table>

Very small errors where the C source was annotated by analyzing the *non*-tagged assembler – not always possible.

Larger errors are due to errors in the matching mechanism (a one-to-one correspondence between the C source and assembler basic blocks is not possible) or influences of the tagging on the compiler optimizations.
Conclusions

- VPM-C
  - Features a high accuracy when simulating the code it was tuned for.
  - The BSS file generation can be automated
  - In case of limited code coverage during the BSS generation phase, it might feature unpredictable accuracy variations when the code or input data changes.
  - The code coverage depends also on the data set used as input to generate the model.
  - Assumes a perfect cache.
  - Requires cycle accurate ISS and target compiler (only by the modeler not by the user of the model)
  - Good for achieving accurate simulations for data dominated flows, whose control flow remains pretty much unchanged with data variations (e.g., MPEG decoding)
  - Development time for a new BSS ranges from 1 day to 1 week. Fine tuning the BSS to improve the accuracy may go up to 1 month, mostly due to extensive simulations
  - Good if not developing extremely time-critical software (e.g. Interrupt Service Routines), or when the precision of SWE is sufficient for the task at hand (e.g., not for final validation after partial integration on an ECU)
  - Good if SW developer is comfortable in using the Microsoft VC++ IDE, rather than the target processor development environment, which may be more familiar to the designer (and more powerful or usable)
SW estimation in VCC

Conclusions

• CABA
  – Fast simulation, comparable with VPM.
  – Good to very good accuracy, since the measurements are based on the real assembler and target architecture effects.
  – Good stability with respect to code or execution flow changes
  – The production target compiler is needed (both modeler and user)
  – About 1 man-month for building a CABA-VI infrastructure, with one processor model.
  – From 2 weeks to 2 months to integrate a new processor – depending upon the simulation time required for the calibration
  – Combines the fast simulation, that characterizes the VPM-based techniques, with the high accuracy of the object code analysis techniques, such as CCISS and ISS integration.
  – Although too few experiments were conducted to know how well it suits various kinds of targets and what is its accuracy and stability to input data and control flow variations, they appear to be promising.
Outline

- SW estimation overview
- Program path analysis
- Micro-architecture modeling
- Implementation examples: Cinderella
- SW estimation in VCC
- SW estimation in AI
- SW estimation in POLIS
SW estimation in AI

• What is AI
  – AI: Abstract Interpretation
  – AI = semantics based methodology for program analyses

• Basic ideas
  – Basic idea of AI: perform the program’s computations using value descriptions or abstract value in place of the concrete values
  – Basic idea of the timing analysis: derive timing information from an approximation of the “collecting semantics” for all inputs

• AI supports correctness proofs
• AI provides tool support (PAG)
Overall Structure

Executable program

CFG Builder

Loop Trafo

CRL File

Static Analyses
- Value Analyzer
- Cache/Pipeline Analyzer

Path Analysis
- ILP-Generator
- LP-Solver
- Evaluation

Worst-case Path Determination

Loop bounds

WCET Visualization

Processor-Behavior Prediction
## Analysis Results (Airbus Benchmark)

<table>
<thead>
<tr>
<th>Task</th>
<th>Airbus’ method</th>
<th>aiT’s results</th>
<th>precision improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.11 ms</td>
<td>5.50 ms</td>
<td>10.0 %</td>
</tr>
<tr>
<td>2</td>
<td>6.29 ms</td>
<td>5.53 ms</td>
<td>12.0 %</td>
</tr>
<tr>
<td>3</td>
<td>6.07 ms</td>
<td>5.48 ms</td>
<td>9.7 %</td>
</tr>
<tr>
<td>4</td>
<td>5.98 ms</td>
<td>5.61 ms</td>
<td>6.2 %</td>
</tr>
<tr>
<td>5</td>
<td>6.05 ms</td>
<td>5.54 ms</td>
<td>8.4 %</td>
</tr>
<tr>
<td>6</td>
<td>6.29 ms</td>
<td>5.49 ms</td>
<td>12.7 %</td>
</tr>
<tr>
<td>7</td>
<td>6.10 ms</td>
<td>5.35 ms</td>
<td>12.3 %</td>
</tr>
<tr>
<td>8</td>
<td>5.99 ms</td>
<td>5.49 ms</td>
<td>8.3 %</td>
</tr>
<tr>
<td>9</td>
<td>6.09 ms</td>
<td>5.45 ms</td>
<td>10.5 %</td>
</tr>
<tr>
<td>10</td>
<td>6.12 ms</td>
<td>5.39 ms</td>
<td>11.9 %</td>
</tr>
<tr>
<td>11</td>
<td>6.00 ms</td>
<td>5.19 ms</td>
<td>13.5 %</td>
</tr>
<tr>
<td>12</td>
<td>5.97 ms</td>
<td>5.40 ms</td>
<td>9.5 %</td>
</tr>
</tbody>
</table>
Interpretation

• Airbus’ results obtained with legacy method: measurement for blocks, tree-based composition, added safety margin
• ~20% overestimation
• aiT’s results were between real worst-case execution times and Airbus’ results
• Compare this with a factor of 30 for PPC 750 with caches switched off!
• Analysis Times for the Benchmark
  – 12 tasks
  – Each ~13 kBytes of instructions
  – Analysis took less than ½ hour per task
The Problem Statement

Given:

- **Processor Timing Model**
  - Abstract processor model
  - Conservative as to the timing behavior
- Fully linked **executable program**
- Annotations (loop and recursion bounds)
- Maybe a **Deadline**
  - Formally it makes a difference
  - In practice it does not; a **sharp upper bound** is required
Cache-Behavior Prediction by Abstract Interpretation
Overall Structure

Executable program

CFG Builder

Loop Trafo

CRL File

Static Analyses

Value Analyzer

Cache/Pipeline Analyzer

Path Analysis

ILP-Generator

LP-Solver

Evaluation

PER File

Processor-Behavior Prediction

Worst-case Path Determination

Loop bounds

WCET Visualization
Caches: Fast Memory on Chip

• Caches are used, because
  – Fast main memory is too expensive
  – The speed gap between CPU and memory is too large and increasing

• Caches work well in the average case:
  – Programs access data locally (many hits)
  – Programs reuse items (instructions, data)
  – Access patterns are distributed evenly across the cache
Caches: How the work

- CPU wants to read/write at memory address a, sends a request for a to the bus

- Cases:
  - hit: Block m containing a is in the cache, request for a is served in the next cycle
  - miss: Block m is not in the cache, m is transferred from main memory to the cache, m may replace some block in the cache, request for a is served asap while transfer still continues

- Several replacement strategies: LRU, PLRU, FIFO,... determine which line to replace
A-Way Set Associative Cache

Address:

Set number

Byte in line

CPU

1

2

... A

Adr. prefix Tag Rep Data block Adr. prefix Tag Rep Data block ... ...

Set: Fully associative subcache of A elements with LRU, FIFO, rand. replacement strategy

... ...

... ...

... ...

... ...

Main Memory

Compare address prefix
If not equal, fetch block from memory

Byte select & align

Data Out
Cache Analysis

• A-way set-associative cache, LRU replacement, $1 \leq i \leq A$.

• Memory block $s$ is in the cache at node $n$ with age at most $i$ iff an access to $s$ appears among the last $i$ different accesses on each path to $n$.

• How to statically precompute cache contents: Must Analysis:
  – for each program point (and context), find out which blocks are in the cache
  – determines safe information about cache hits. Each predicted cache hit reduces WCET.
Cache with LRU Replacement: Transfer

concrete

abstract
Cache Analysis: Join (must)

Join (must)

```
{ a }
{ }   
{ c, f }
{ d }

{ c }
{ e }
{ a }
{ d }
```

"intersection + maximal age"
Cache Analysis: Join (may)

\[ \{a\} \]
\[ \{c, f\} \]
\[ \{\} \]
\[ \{d\} \]
\[ \{c\} \]
\[ \{e\} \]
\[ \{a\} \]
\[ \{d\} \]

"union + minimal age"

Question: How many references will a memory block maximally survive in the cache?
SW estimation in AI

- Cache analysis (cont.)

Approximation of the Collecting Semantics

- the semantics determines set of all cache states for each program point
- “cache” semantics determines set of all cache states for each program point
- abstract semantics determines abstract cache states for each program point
Pipeline Analysis

- Integrates all the components: Cache, bus, memory subsystem
- Abstract model obtained by reduction and abstraction
Abstract Instruction-Execution

mul

Fetch
I-Cache miss?

Issue
Unit occupied?

Execute
Multicycle?

Retire
Pending instructions?

s#

unknown
Constant Propagation by AI and MC

• CP determines for each program point $n$ a set of variable-value pairs $(x, c)$ such that $x$ has value $c$ each time execution reaches $n$.

• MC can verify that $x$ has value $c$ if you tell it $c$.

• MC can find out that $x$ has value $c$ if you let MC run through the whole domain of values.

• Principled argument for infinite domains.

• Practicality argument for finite, but large domains.
Substitute for AI?

Under different definitions of MC

• **Reachability**: MC is fixed point iteration *without dynamic abstraction* using *set union* to collect states:
  Needs to search an exponentially large state space, e.g. potential cache-set contents for *a-way* set-associative set, $k = \#\text{memory blocks mapped to the set}$

• In contrast: AI is fixed point iteration *with dynamic abstraction* using *join* to combine states
Cache-Behavior Prediction

• Must-cache analysis (AI) determines for each program point \( n \) a set of memory blocks \( s \) such that
  \( s \) is in the cache each time execution reaches \( n \)

• MC can verify that \( s \) is in the cache if you tell it \( s \)

• MC can find out that \( s \) is in the cache if you let MC run through the whole domain of cache states

• But we know \( s \)! Memory blocks referenced at \( n \)
A static analysis may verify several safety properties at each program point in one fixed point iteration

- Cache hits, pipeline advances, etc.

• Individual safety properties need not be specified! 😊
• They are encoded in the static analysis

What is the substitute for this power on the MC side?
More definitions of MC

• MC checks a specified property of the system:
  – Local: safety properties at program points
  – Global: One global property of the program

• MC = AI: 😞
MC – How are we doing?

• **Local** safety properties, e.g. s is in cache, can be model checked

• What is the **complexity**?

• Exponential in A – no problem!

• Influence of pipeline, branch prediction on (instruction) cache ⇒ Exponential in pipeline and branch-prediction parameters – not a big problem!
MC – How are we doing?

- How many runs of MC to check local properties?
- Is $O(n)$ enough for a program of size $n$?
- What about (nested) loops?
- **Contexts**, i.e. different combinations of iterations, are important for precision
- In **MC**: increases #runs
- In **AI**: extends the one fixed point iteration, easily fine tuned
Model Check one Global Formula

- Conjunction over local formulae
- Local formulae are disjunctions over all possibilities to execute the instruction
- Needs the same effort as the Reachability approach
Once again: The Problem Statement

Given:

• **Processor Timing Model**
  – Abstract processor model
  – Conservative as to the timing behavior

• Fully linked **executable program**

• Annotations (loop and recursion bounds)

• Maybe a **Deadline**
  – Formally it makes a difference
  – In practice it does not; a **sharp upper bound** is required
Comparing Apples with Oranges

• Complaints (Ken McMillan):
  – AI is losing information through abstraction
  – MC has complete information

• Customers cannot wait arbitrarily long for complete information

• Experience shows that AI is not losing much information
WCET Determination by AI + ILP

• WCET determination using AI + ILP
  – Yields precise results
  – Runs at acceptable speed

“Usaar’s and AbsInt’s work on WCET determination is considered a real breakthrough, and AbsInt’s tool is probably the best of its kind in the world”
Report of Final Review of the IST project Daedalus

• Complexity arguments against MC
• One attempt used ILP, also had complexity problems
References

- R. Ernst, W. Ye, *Embedded program timing analysis based on path clustering and architecture classification*, ICCAD 1997
- S. Malik, M. Martonosi, and Y. T. S. Li, *Newblock Static timing analysis of embedded software*, DAC 1997
- P. Cousot, R. Cousot, *Verification of Embedded Software: Problems and Perspectives*, EMSOFT 2001
Outline

- SW estimation overview
- Program path analysis
- Micro-architecture modeling
- Implementation examples: Cinderella
- SW estimation in VCC
- SW estimation in AI
- SW estimation in POLIS
POLIS Methodology

Aptix Board Consist of
- micro of choice
- FPGA’s
- FPIC’s

Graphical EFSM+
Esterel

Java

EC

..............

Compilers

Sw Synthesis

CFSMs

Sw Estimation

Partitioning

Hw Estimation

Hw Synthesis

Hw/Sw Co-Simulation
Performance/trade-off Evaluation
Fomal Verification

Sw Code +
RTOS

Physical Prototyping

Logic Netlist

Aptix Board Consist of
- micro of choice
- FPGA’s
- FPIC’s
Hardware - Software Architecture

• Hardware:
  – currently:
    – Programmable processors (micro-controllers, DSPs)
    – ASICs (FPGAs)

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

• Interfaces:
  – Hardware modules
  – Software procedures (polling, interrupt handlers, ...)

EE249Fall04
System Partitioning

[Diagram showing system partitioning with nodes labeled CFSM1, CFSM2, CFSM3, CFSM4, CFSM5, CFSM6, CFSM7, and Scheduler. Edges labeled e1, e2, e3, e5, e7, and e8 connect the nodes, indicating transitions between states.}
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
    \[ f = x f_x + x f_x \]
  - 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
      \( f = \bar{x}_1 + x_2 x_3 \)

**Diagram:**

- ROBDD

- 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
Technique For Handling ROBDD Memory Explosion

All the representations are canonical for combinational equivalence checking.
Handling Memory Explosion: Variable Ordering

- BDD size very sensitive to variable ordering

\[ a_1 b_1 + a_2 b_2 + a_3 b_3 \]

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

\[ a_1 b_1 + a_2 b_2 \]
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)
  - 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 sub-expression extraction)
  - map expression DAGs to C expressions
  - C compiler allocates registers and select op-codes
- 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

Flowchart:
- Specification, partitioning
  - S-graph synthesis
    - Timing estimation
      - Scheduling, validation
        - feasible
          - Code generation
          - Compilation
          - Testing, validation
            - pass
            - Production
        - not feasible
          - Flow back to specification, partitioning
SW estimation in POLIS

**S-graph Level Estimation**

- CFSM
- SW synthesis
- S-graph synthesis and optimization
- **Estimation**
  - Timing / code size information
- S-graph
- Code generation
- Sw code
SW estimation in POLIS

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

```
func(E)
event E;
{
    static int st;
    Initialization of local variables;
    Structure of mixed if or switch statements and assign statements;
    return;
}
```

generated C code

Time $T$ and Size $S$

\[
T = T_{pp} + k T_{init} + T_{struct}
\]

\[
S = S_{pp} + k S_{init} + S_{struct}
\]
Execution time of a path and code size

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

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

- \( \pi_i \): 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 \( v \).

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

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

Path on S-graph
SW estimation in POLIS

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) $T_{init}$, $S_{init}$: Execution time and code size for local variable initialization.
SW estimation in POLIS

• Problems II

  – How to handle the variety of compilers and CPUs?

    – prepare cost parameters for each target
**Extraction of Cost Parameters**

- set of benchmark programs
- target C compiler
- static analyzer
- execution & profiling
- parameter extractor
- cost parameters

SW estimation in POLIS
**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.
Cost $C$ is a triple $(\text{min\_time}, \text{max\_time}, \text{code\_size})$

Algorithm: $SGtrace\ (sg_i)$
   if ($sg_i == \text{NULL}$) return $(C(0, ,0))$;
   if ($sg_i$ has been visited)
      return (pre-calculated $C_i(*,*,0)$ associated with $sg_i$);
   $C_i = \text{initialize} \ (\text{max\_time} = 0, \text{min\_time} = , \text{code\_size} = 0)$
   for each child $sg_j$ of $sg_i$
      \[
      C_{ij} = SGtrace\ (sg_j) + \text{edge cost for edge } e_{ij};
      \]
      $C_i.\text{max\_time} = \max(C_i.\text{max\_time}, C_{ij}.\text{max\_time});$
      $C_i.\text{min\_time} = \min(C_i.\text{min\_time}, C_{ij}.\text{min\_time});$
      \[
      C_i.\text{code\_size} += C_{ij}.\text{code\_size};
      \]
   $C_i += \text{node cost for node } sg_i;$
   return $(C_i)$;
SW estimation in POLIS

Experiments

* Proposed methods implemented and examined in POLIS system.

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

* Difference D is defined as

\[ D = \frac{\text{cost}_{\text{estimated}} - \text{cost}_{\text{measured}}}{\text{cost}_{\text{measured}}} \]
## Experimental Results: S-graph Level

<table>
<thead>
<tr>
<th>model</th>
<th>estimated</th>
<th>measured</th>
<th>% difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>158</td>
<td>141</td>
<td>12.06</td>
</tr>
<tr>
<td>max</td>
<td>469</td>
<td>496</td>
<td>-5.44</td>
</tr>
<tr>
<td>size</td>
<td>654</td>
<td>690</td>
<td>5.22</td>
</tr>
<tr>
<td>TIMER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>223</td>
<td>191</td>
<td>16.75</td>
</tr>
<tr>
<td>max</td>
<td>938</td>
<td>912</td>
<td>2.85</td>
</tr>
<tr>
<td>size</td>
<td>1,573</td>
<td>1,436</td>
<td>9.54</td>
</tr>
<tr>
<td>ODOMETER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>145</td>
<td>131</td>
<td>10.69</td>
</tr>
<tr>
<td>max</td>
<td>361</td>
<td>363</td>
<td>-0.55</td>
</tr>
<tr>
<td>size</td>
<td>454</td>
<td>457</td>
<td>-0.66</td>
</tr>
<tr>
<td>SPEEDOMETER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>314</td>
<td>335</td>
<td>-6.27</td>
</tr>
<tr>
<td>max</td>
<td>880</td>
<td>969</td>
<td>-9.18</td>
</tr>
<tr>
<td>size</td>
<td>764</td>
<td>838</td>
<td>-8.83</td>
</tr>
<tr>
<td>BELT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>119</td>
<td>111</td>
<td>7.21</td>
</tr>
<tr>
<td>max</td>
<td>322</td>
<td>323</td>
<td>-0.31</td>
</tr>
<tr>
<td>size</td>
<td>511</td>
<td>520</td>
<td>-1.73</td>
</tr>
<tr>
<td>FUEL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>197</td>
<td>171</td>
<td>15.20</td>
</tr>
<tr>
<td>max</td>
<td>533</td>
<td>586</td>
<td>-9.04</td>
</tr>
<tr>
<td>size</td>
<td>637</td>
<td>647</td>
<td>-1.55</td>
</tr>
<tr>
<td>CROSSDISP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>262</td>
<td>221</td>
<td>18.55</td>
</tr>
<tr>
<td>max</td>
<td>16,289</td>
<td>16,979</td>
<td>-4.06</td>
</tr>
<tr>
<td>size</td>
<td>32,592</td>
<td>38,618</td>
<td>-15.60</td>
</tr>
</tbody>
</table>
SW estimation in POLIS

Performance and cost estimation: example

• 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
SW estimation in POLIS

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