Software Performance Estimation by Static Analysis

Presented by: Robert Nilsson

References are available at the hw/sw co-design course web page
Outline

- Introduction and problem description
- Basic WCET estimation techniques
- Path Clustering and architecture classification
  - SFP / MFP - analysis
- Instruction cache modeling
  - direct memory mapped caches
- Pipelined architectures?
Introduction

- Real-time systems need to guarantee timeliness

- Scheduling analysis assume known wcet of tasks
  - to predict worst case response times
  - also bcet (best-case execution times) relevant in some systems

- Useful during hw/sw codesign decision to map functions to resources
Introduction

- Worst case execution time analysis must be conservative
  - correctness (safety) vs. resource usage issue

- The execution time of a program depends on
  - program path
    - data dependent
  - computer architecture properties
    - e.g. caches, pipelines, etc
Problems

- Program path analysis is in general an undecidable problem
  - E.g. unbounded loops, recursive function calls
  - Even if such constructs are prohibited the number of program paths grows aggressively with
    - nested loops
    - branches in loops

- Difficult to model complex microarchitectures
  - E.g. caches, branch prediction, superscalar processors
  - Too expensive to neglect the impact of hardware properties
    - very pessimistic estimations → large resource waste
Program path analysis techniques

- Program code is classified into basic blocks
  - a program segment which is only entered at the first statement and only left at the last statement

- The basic blocks represents nodes in the program flow graph

- program paths are determined by traversing the flow graph from start block to end block
  - feasible paths can occur during execution
  - false paths can never occur during execution

- We need clever ways to classify paths!
Program path classification techniques

- loop annotation required for all loops
  - puts a bound on the number of program paths
- manually identification of classes of “false paths”
- implicit path enumeration
  - linear equations are used to specify constraints between blocks execution
    - e.g. $x_1 \leq x_2$ means block $x_1$ is executed at most as often as $x_2$
  → integer linear programming optimization problem
    - e.g. simplex algorithm
Basic block timing analysis

The actual execution times for a basic block is acquired trough:

- **Instruction timing addition (ITA)**
  - execution time of instructions are added
    - number of processor cycles for each instruction

- **Path segment simulation (PSS)**
  - a cycle true processor model is used to simulate the execution of the basic block (or a path segment)
Architectural properties and time analysis

- **Data dependent instruction execution times**
  - hard for simulators to guarantee accurate timing
  - ITA could be used (using worst-case)

- **Pipelined architectures**
  - PSS must assume worst-case behavior on block boundaries
  - ITA cannot anticipate pipeline hazards

- **Superscalar architectures**
  - ITA is completely inappropriate if it does not model instruction scheduling
Architectural properties and time analysis, cnt.

- **Program (instruction) caches**
  - PSS can be exact for program caches since the cache is simulated
    - worst case must be assumed at block boundaries
    - Better estimations if the blocks are big
  - ITA does not cover caches
    - need a method to model cache-hits and cache misses

- **Data caches**
  - PSS is precise if the same data variables always are accessed in a block
  - Same properties as program caches
Execution time models

- Simplest is **sum-of-basic-blocks** model
  - each basic block has an associated constant execution time

- In many architectures this is not realistic
  - data dependent instruction execution times
  - overlapped basic block execution
  - ...
    ➔ Different sequences of basic blocks (**path segments**) gives different execution times
      • **sequences-of-basic-blocks** model needed
Program partitioning

- Analysis of **sequences-of-basic-blocks** model require exhaustive path analysis in the worst case →we’re in trouble!

- Analysis is simplified if the program path is independent of input data
  - always get the same sequence of basic blocks
  - common in e.g. signaling processing applications
  - **single feasible path (SFP)** property

- However, most practical programs have at least some parts where the program path depends on input variables
  - **multiple feasible paths (MFP)** property
Symbolic hybrid timing analysis method

- R. Ernst, W. Ye (1997)

- Aim to partition the program in parts with the SFP and MFP properties

- “Hierarchal flow graph clustering” can be used for the partitioning
  - control constructs are hierarchical nodes
  - basic blocks are leaf nodes
    - Basic blocks are SFP by definition
    - A hierarchical node is SFP if
      - it only contain SFP nodes
      - its associated condition is independent of input data
Hierarchical flow graph clustering example

for (i=0; i<14; i++)
for (j=i+1; j<15; j++)
    if (a[i]<a[j])
    {
        tmp=a[i];
        a[i]=a[j];
        a[j]=tmp;
    }

Cut Point
Global timing analysis

- For each SFP block, the execution time is determined.
  - using PSS or ITA as appropriate for the architecture
  - assuming worst-case behaviors at “cut-points”

- The resulting execution times for the SFP blocks are summarized

- For the remaining MFP parts, worst-case behavior is acquired
  - using for example an Integer Linear Programming approach
  - a pessimistic constant cost for each basic block in the reduced flow graphs
Experiments

- A tool was implemented for supporting the SYMTA approach
- Set of example programs was used
- Two architectures
  - superscalar SPARC with 4-stage pipelines
  - Intel 8051
## Experimental results

<table>
<thead>
<tr>
<th>Programs</th>
<th>Total nodes</th>
<th>Nodes in SFP</th>
<th>Nodes in MFP</th>
<th>Source lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D-image</td>
<td>94</td>
<td>85</td>
<td>90%</td>
<td>164</td>
</tr>
<tr>
<td>diesel</td>
<td>65</td>
<td>65</td>
<td>100%</td>
<td>160</td>
</tr>
<tr>
<td>fft</td>
<td>78</td>
<td>78</td>
<td>100%</td>
<td>145</td>
</tr>
<tr>
<td>bsort</td>
<td>14</td>
<td>8</td>
<td>57%</td>
<td>25</td>
</tr>
<tr>
<td>smooth</td>
<td>48</td>
<td>39</td>
<td>81%</td>
<td>86</td>
</tr>
<tr>
<td>blue</td>
<td>80</td>
<td>53</td>
<td>66%</td>
<td>127</td>
</tr>
<tr>
<td>check-data</td>
<td>18</td>
<td>0</td>
<td>0%</td>
<td>44</td>
</tr>
<tr>
<td>whetstone</td>
<td>122</td>
<td>122</td>
<td>100%</td>
<td>251</td>
</tr>
<tr>
<td>line</td>
<td>101</td>
<td>19</td>
<td>19%</td>
<td>250</td>
</tr>
<tr>
<td>key3</td>
<td>100</td>
<td>100</td>
<td>100%</td>
<td>151</td>
</tr>
</tbody>
</table>

Table 1: Experimental results for the Clustering
Experimental results

<table>
<thead>
<tr>
<th>Programs</th>
<th>Measured bounds (cycles)</th>
<th>Analyzed bounds (cycles)</th>
<th>Analysis time* (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BCET**</td>
<td>WCET**</td>
<td>BCET</td>
</tr>
<tr>
<td>SPARC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D-image</td>
<td>34908</td>
<td>37848</td>
<td>33874</td>
</tr>
<tr>
<td>diesel</td>
<td>62944</td>
<td>62994</td>
<td>61445</td>
</tr>
<tr>
<td>fft</td>
<td>1498817</td>
<td>1499176</td>
<td>1494650</td>
</tr>
<tr>
<td>bsort</td>
<td>4423</td>
<td>8938</td>
<td>4423</td>
</tr>
<tr>
<td>smooth</td>
<td>3635651</td>
<td>4846511</td>
<td>3570227</td>
</tr>
<tr>
<td>blue</td>
<td>3564938</td>
<td>316865761</td>
<td>3345041</td>
</tr>
<tr>
<td>check-data</td>
<td>80</td>
<td>431</td>
<td>65</td>
</tr>
<tr>
<td>whetstone</td>
<td>2928459</td>
<td>3369459</td>
<td>2880230</td>
</tr>
<tr>
<td>line</td>
<td>514</td>
<td>1619</td>
<td>381</td>
</tr>
<tr>
<td>8051</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fft</td>
<td>26421460</td>
<td>26421460</td>
<td>26419338</td>
</tr>
<tr>
<td>bsort</td>
<td>9347</td>
<td>15045</td>
<td>7804</td>
</tr>
<tr>
<td>smooth</td>
<td>9737378</td>
<td>9737516</td>
<td>9737469</td>
</tr>
<tr>
<td>key3</td>
<td>1218229</td>
<td>1223314</td>
<td>1164883</td>
</tr>
<tr>
<td>check-data</td>
<td>68</td>
<td>559</td>
<td>63</td>
</tr>
</tbody>
</table>

* The example programs have been analyzed on the SPARC 10 workstation.
** WCET: The worst case execution time; BCET: The best case execution time.

Table 2: Experimental results of the example programs in SYMTA
Instruction Cache modeling

- A method for getting tighter time estimations of programs running on architectures with instruction caches
- Cache memories are difficult to model and impose a lot of pessimism if neglected
  - direct mapped-instruction caches
Program path analysis as ILP

- Program path analysis can be transformed into a ILP problem using
  - program structural constraints
    - derived from program control flow graph (CFG)
  - program functionality constraints
    - provided by the user
    - e.g. specifies loop bounds

- *Without cache modeling a constant instruction execution time is assumed, hence;*

\[
\text{Execution time} = \sum_{i=1}^{N} c_i x_i
\]

\[c = \text{execution time for basic block}\]
\[x = \text{execution count for basic block}\]
Program path analysis example

- structural constraints
  
  \[ d_1 = 1 \]
  \[ x_1 = d_1 = d_2 \]
  \[ x_2 = d_2 + d_8 = d_3 + d_9 \]
  \[ x_3 = d_3 = d_4 + d_5 \]
  \[ x_4 = d_4 = d_6 \]
  \[ x_5 = d_5 = d_7 \]
  \[ x_6 = d_6 + d_7 = d_8 \]
  \[ x_7 = d_9 = d_{10} \]
Program path analysis example

- functional constraints
  
  \[0 \times 1 \leq x_3 \leq 10 \times 1\]

  \[x_5 \leq 1 \times 1\]

  ...

- All these constraints are passed to the ILP-solver
Direct mapped Instruction caches

- the code in basic blocks are divided into a number of line-blocks
- the line blocks are assigned to cache lines
  - cache sets (cache lines) represent physical cache memory
Adding cache analysis to the ILP model

- Now execution times of basic blocks differ if the line-blocks are in the cache or not

\[
\text{Execution\_time} = \sum_{i=1}^{N} \sum_{j=1}^{n_i} (c_{i,j} x_{i,j}^{\text{hit}} + c_{i,j} x_{i,j}^{\text{miss}})
\]

- The execution count of a basic block becomes

\[
x_i = x_{i,j}^{\text{hit}} + x_{i,j}^{\text{miss}}
\]

\[j = 1,2... n_i\]

\[i = \text{all basic blocks}\]

\[j = \text{all line blocks in block } i\]

\[x_{i,j}^{\text{hit}} = \text{number of cache hits}\]

\[x_{i,j}^{\text{miss}} = \text{number of cache misses}\]

\[c_{i,j}^{\text{hit}} = \text{execution time for cache hit}\]

\[c_{i,j}^{\text{miss}} = \text{execution time of cache miss}\]
There are three possible types of cache assignments that can occur

- Only one line-block assigned to a cache line
  - when a miss occurs, the line-block will be loaded and no more cache misses will occur
    \[ x_{k.l}^{miss} \leq 1 \]

- Two or more *nonconflicting* line-blocks are assigned to the same cache line
  - when a miss occurs in either block, the line-blocks will be loaded and no more cache misses will occur
    \[ x_{1.3}^{miss} + x_{2.1}^{miss} \leq 1 \]

- a cache line contains two or more conflicting line-blocks
Cache conflict graphs

- s and e nodes represent the start and the end of the program respectively.
- B – nodes represent conflicting line-blocks.
- Edges represent possible program flow between blocks – acquired from program cfg.
- \( p(\text{node1, node2}) \) is a counter associated with each edge.
Constraints on cache conflict graphs

- The counters \((p)\) are bound to the structural and functional constraints through the \(x\) variables
  - the execution count of a line-block must be equal to the execution count of the basic block
  - the control flow to a line-block node must be equal to the flow from the line-block node

\[
x_i = \sum_{u,v} p(u,v,i,j) = \sum_{u,v} p(i,j,u,v)
\]
Loops and cache constraints

\[ p(s, 7.1) + p(4.1) \leq x_5 \]
Experiments

- Intel QT960
  - 32 x 16 bytes direct-mapped instruction cache

- Execution times are assigned statically to line-blocks hits and misses

- tool, called “cinderella” was implemented
  - generate CFG
  - generates CCG
  - output structural constraints
  - ask user for loop bounds and other optimization constraints
  - used in conjunction with public domain ILP-solver

- Evaluation programs chosen so that worst-case input data was available for measurement and comparison


Experiments results

<table>
<thead>
<tr>
<th>Program</th>
<th>Measured WCET</th>
<th>Estimated WCET</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>check data</td>
<td>$4.30 \times 10^2$</td>
<td>$4.91 \times 10^2$</td>
<td>1.14</td>
</tr>
<tr>
<td>circle</td>
<td>$1.45 \times 10^4$</td>
<td>$1.54 \times 10^4$</td>
<td>1.06</td>
</tr>
<tr>
<td>des</td>
<td>$2.44 \times 10^5$</td>
<td>$3.70 \times 10^5$</td>
<td>1.52</td>
</tr>
<tr>
<td>dhry</td>
<td>$5.76 \times 10^5$</td>
<td>$7.57 \times 10^5$</td>
<td>1.31</td>
</tr>
<tr>
<td>djpeg</td>
<td>$3.56 \times 10^7$</td>
<td>$7.04 \times 10^7$</td>
<td>1.98</td>
</tr>
<tr>
<td>fdct</td>
<td>$9.05 \times 10^3$</td>
<td>$9.11 \times 10^3$</td>
<td>1.01</td>
</tr>
<tr>
<td>fft</td>
<td>$2.20 \times 10^6$</td>
<td>$2.63 \times 10^6$</td>
<td>1.20</td>
</tr>
<tr>
<td>line</td>
<td>$4.84 \times 10^3$</td>
<td>$6.09 \times 10^3$</td>
<td>1.26</td>
</tr>
<tr>
<td>matcnt</td>
<td>$2.20 \times 10^6$</td>
<td>$5.46 \times 10^6$</td>
<td>2.48</td>
</tr>
<tr>
<td>matcnt2</td>
<td>$1.86 \times 10^6$</td>
<td>$2.11 \times 10^6$</td>
<td>1.13</td>
</tr>
<tr>
<td>pkrts</td>
<td>$1.71 \times 10^3$</td>
<td>$1.74 \times 10^3$</td>
<td>1.02</td>
</tr>
<tr>
<td>sort</td>
<td>$9.99 \times 10^6$</td>
<td>$27.8 \times 10^6$</td>
<td>2.78</td>
</tr>
<tr>
<td>sort2</td>
<td>$6.75 \times 10^6$</td>
<td>$7.09 \times 10^6$</td>
<td>1.05</td>
</tr>
<tr>
<td>stats</td>
<td>$1.16 \times 10^6$</td>
<td>$2.21 \times 10^6$</td>
<td>1.91</td>
</tr>
<tr>
<td>stats2</td>
<td>$1.06 \times 10^6$</td>
<td>$1.24 \times 10^6$</td>
<td>1.17</td>
</tr>
<tr>
<td>whetstone</td>
<td>$6.94 \times 10^6$</td>
<td>$10.5 \times 10^6$</td>
<td>1.51</td>
</tr>
</tbody>
</table>
WCET analysis beyond direct mapped instruction caches

- There are several variables which influence the complexity of cache analysis
  - number of competing line-blocks \( (m) \)
  - cache associativity level \( (n) \)
  - cache replacement method

- For LRU (least recently used), the complexity grows as

\[
\sum_{i=0}^{n} \frac{m!}{(m-i)!}
\]

- By using a more detailed level of cache modeling better estimations can be acquired, but the problem becomes intractable if programs are large
Some measurements
Analysis of multi-issue Pipelines

- Previously we have assumed that no parallelism occur in the processor

- If we can model the gain from instruction level parallelism we can get a tighter WCET bound

- When introducing pipelines in our model, the execution time depends on instruction scheduling
  - the code provides data-dependencies and order between instructions
  - the “assignment” of instruction types to resource types needs to be explicitly given
Modeling instruction level schedules

- A processor is said to have a set of resource types. For each resource type, there are a set of instances of this resource type – e.g., 2 floating point units.

- The program consists of a queue of instructions (in basic block or line-block)
  - The priority of the instruction is set according to the order of the program and data-dependencies.
  - If two instructions have no data-dependencies, they can have the same priority.
Pipelined architecture
Simplified scheduling algorithm

For each time unit
  – for each resource type
    • Determine set of instructions which can be executed at this time-unit without violating data dependencies
    • Determine which resource instances that can execute instructions of this type at this time
    • assign highest priority instructions to free resources
  – Continue until all instructions are scheduled
Prediction gain

The schedule gives the number of time units that are required for executing the block (or the line-block if caches are used)
Summary

- Performance estimations neglecting processor architectures are to pessimistic
- Avoid explicit enumeration of execution paths
  - Program partitioning
  - implicit path enumeration $\rightarrow$ ILP problem
- Cache modeling improve tightness considerable
  - but for advanced cache models the complexity of analysis explode
- Pipeline schedule modeling
  - Improves bound slightly
  - Knowledge about instruction scheduling policy required