Execution time evaluation

Prof. J.-D. Decotignie
CSEM Centre Suisse d’Electronique et de Microtechnique SA
Jaquet-Droz 1, 2007 Neuchâtel
jean-dominique.decotignie@csem.ch

Plan

- Introduction
- Parameters
- Measurement
- Simulation
- Calculation

Introduction

- It is necessary to know the WCET
  - The best case may also be interesting
  - Different approaches
    - The target and the code are available
    - The code is available but not the target

Impacting parameters

- Related to the code
  - compiler
  - Loop limiting
  - Program structure
- Related to hardware
  - Speed and type of processor
  - Cache memories
  - pipeline
  - Speed and type of memory
  - disks
  - ...
### Program structure

FOR I:=1 TO N DO
BEGIN
  IF I<5 THEN
    X[I] := X[I-1] + 3;
  ELSE
    X[I] := X[I-1] * 5;
END;

- Difficult to predict N
  - limit the number of iterations
  - no recursion, no GOTO
- Numerous functionality
- Source code is necessary

### Compilation

- The generated code quality has some influence
- It is necessary to have the compiled code (assembly)

FOR I:=1 TO N DO
BEGIN
  IF I<5 THEN
    X[I] := X[I-1] + 3;
  ELSE
    X[I] := X[I-1] * 5;
END;

### Hardware

- Memory access time
- Processor speed
- Data dependencies (multiplication, shift)
- Pipeline

### Pipeline

- MC 68010
  - MOVE.W #Length, D0
  - LOOP: MOVE.W (A0)+,(A1)+
  - DBEQ DO, LOOP

- 14 levels on Pentium Pro
- Problems
  - data => temporary stall
  - control => stop or purge
  - Loop prediction
Cache memories

- Often not welcome
- However performances are so much better
- cache miss = 24 cycles on SPARC 2
- Problem in case of context switch
  => can be partitioned
- How to predict their behavior

Memories

- Dynamic memories must be refreshed periodically
- The access time of the first byte is longer than the next ones for fast memories (SDRAM)
  => access by bursts (blocks)

Memory management

- Logical addressing (or virtual)
  => translation
- Translation tables with cache for the most recent ones
- Virtual memory
  => disk access time should be taken into account

Degrees of influence

<table>
<thead>
<tr>
<th>Feature</th>
<th>ratio worst/best</th>
<th>mean/best</th>
<th>ratio worst/mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loading data cache</td>
<td>[14.5, 8.0]</td>
<td>[1.5, 1.3]</td>
<td>[9.7, 6.2]</td>
</tr>
<tr>
<td>Pipeline effects</td>
<td>12</td>
<td>Not available</td>
<td>Not available</td>
</tr>
<tr>
<td>Loading instruction cache</td>
<td>7</td>
<td>1.8</td>
<td>3.9</td>
</tr>
<tr>
<td>Writing data cache</td>
<td>2</td>
<td>1.5</td>
<td>1.3</td>
</tr>
<tr>
<td>jumps</td>
<td>1.9</td>
<td>1.1</td>
<td>1.7</td>
</tr>
<tr>
<td>ALU instructions</td>
<td>1.8</td>
<td>1.4</td>
<td>1.3</td>
</tr>
<tr>
<td>Exceptions</td>
<td>1.2</td>
<td>1.0</td>
<td>1.2</td>
</tr>
<tr>
<td>refreshing dynamic RAM</td>
<td>1.1</td>
<td>1.0</td>
<td>1.1</td>
</tr>
<tr>
<td>Memory management tables (TLB)</td>
<td>&lt; 88</td>
<td>3.6</td>
<td>&lt;25</td>
</tr>
</tbody>
</table>
Measurement

- Direct measurement
- Double loop technique
- SW stamps
- Indirect measurements

Direct measurement

- Based on a logic analyser

![Timing Waveform Diagram]

- Code may be disassembled by the logic analyzer

![Disassembled Code Table]

- One has to find the correspondence between the high level language code and the assembly language
- Beware of the false triggers

- Connection is sometimes difficult
  - Connect the logic analyzer on other ICs
  - Indirect observation (penalty)
  - Sometimes, it is impossible to observe (caches)
Software probes

- Close to indirect measurements using a logic analyzer
- Clock reading instructions are inserted in the code
- The clock value is recorded (variables, memory, ...)
- A posteriori analysis

- Introduces perturbations in the measurements
- Precision and resolutions are sometimes limited (not on Pentium)
- However, cheap and easy to deploy

Double loop test

- Used when the internal clock has limited resolution

```plaintext
... GET_TIME(T1); (* start instant *)
FOR i:=1 TO n DO
   (* code for which we want to measure the exec. time *)
   END;
GET_TIME(T2); (* end instant *)
GET_TIME(T3); (* start instant *)
FOR i:=1 TO n DO
   END;
GET_TIME(T4); (* end instant *)
....
MeanExecutionTime := ((T2-T1)-(T4-T3))/n;
```

Double loop test

- The measured code must be isolated from its context
  => interactions with it should be emulated
- The execution time may vary depending on the presence or absence of the loop (12%)
- Source code may be compiled differently outside its context

Measurements - Analysis

- Simple techniques
- The hardware should be available
- Often intrusive
- One measurement only reflects one execution is it the worst (best) case?

```plaintext
FOR i:=1 TO n DO
BEGIN
   IF i<5 THEN
      x[i] := x[i-1] + 3;
   ELSE
      x[i] := x[i-1] * 5;
   END;
END;
```
Measurements – Analysis (2)

- Is certainly usable!
- With caches, jumps, pipelines, …

IT BECOMES DIFFICULT TO MEASURE

Simulation

- When the hardware is not available
- Requires detailed informations on the processor operations and its the operations of its peripherals (often confidential)
- The results are highly dependent on the model quality
- Slow
- Problems with the external events (must be emulated)
  => replaced by precedence relationships
- Suffers from the same problems as measurements
  - Are we sure it is the worst (best) case

Calculation

- A whole class of techniques
- Hardware must not be available
- Similar to simulation but we are only interested in the calculation time
  => may be quicker
- The various techniques do not all take into account all aspects (pipeline, cache, memory management, ...)
  - Old techniques => simple processors
  - Still a very active research domain

- Table lookup
- Instruction counting
- Time schemas
- Taking pipelines into account
- Taking caches into account
**Table lookup**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Size</th>
<th>Register</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR, ASL</td>
<td>byte, word long</td>
<td>6 + 2n(1/0)</td>
<td>8(1,1)+</td>
</tr>
<tr>
<td>LSR, LSL</td>
<td>byte, word long</td>
<td>6 + 2n(1/0)</td>
<td>8(1,1)+</td>
</tr>
<tr>
<td>ROR, ROL</td>
<td>byte, word long</td>
<td>6 + 2n(1/0)</td>
<td>8(1,1)+</td>
</tr>
<tr>
<td>ROXL, ROXR</td>
<td>byte, word long</td>
<td>6 + 2n(1/0)</td>
<td>8(1,1)+</td>
</tr>
</tbody>
</table>

**Mode d'adressage**

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Byte, Word</th>
<th>Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>On</td>
<td>0(0/0)</td>
<td>0(0/0)</td>
</tr>
<tr>
<td>An</td>
<td>0(0/0)</td>
<td>0(0/0)</td>
</tr>
<tr>
<td>(An)</td>
<td>4(1/0)</td>
<td>8(2/0)</td>
</tr>
<tr>
<td>-(An)</td>
<td>4(1/0)</td>
<td>8(2/0)</td>
</tr>
<tr>
<td>d16(An)</td>
<td>8(2/0)</td>
<td>12(3/0)</td>
</tr>
</tbody>
</table>

**Instruction counting**

- We count the number of instructions
- The result is multiplied by a constant to get the worst case and another to get the best case execution time
- Constants are determined by measuring the execution of a representative set of programs
- Simple
- Acceptable results

**Time schemas**

- due to Alan Shaw
- The starting point is the high level language program
- 4 steps
  - decomposition of a high level language statement in its basic blocks (defined in its time schema)
  - Prediction of the assembly language implementation of the basic blocks
  - Calculation of the execution time of the basic block from the generated assembly code
  - Calculation of the execution time of the instruction from the basic block execution times and the time schema
**Time schemas - example**

- **Execution time**: \( T(S) = [t_{\text{min}}(S), t_{\text{max}}(S)] \)
  1) time schema of S1: \( a := b + c \) : \( T(S1) = T(b) + T(+) + T(c) + T(a) + T(=) \) => 5 basic blocks.

Addition on intervals: \([t_1, t_2] + [t_3, t_4] = [t_1 + t_3, t_2 + t_4]\)

- **Prediction of assembly language for each block**
  - \( b : \) MOVE M,R \( M \) and \( R \) represent resp.
  - \( + : \) ADD M,R memory locations
  - \( c : \) nothing and generic registers.
  - \( a : \) nothing
  - \( := \) MOVE R,M

3 et 4) according to the time schema, S1 execution time will the sum of the durations of the 3 blocks.

**Control structures**

- \( S : \) if (exp) then S1 else S2 end; 
  \( T(S) = \min(t_{\text{low}}, t_{\text{up}}) \max(t_{\text{up}}, t_{\text{low}}) \)

- with 
  \([t_{1, \text{low}}, t_{1, \text{up}}] = T(\text{exp}) + T(S1) + T(\text{then}) \)

  \([t_{2, \text{low}}, t_{2, \text{up}}] = T(\text{exp}) + T(S2) + T(\text{else}) \)

**Basic block granularity**

- Short blocks (terminal symbol of the language)
  - Predicting is simple if code generation follows the parsing structure (often this is the case only for control structures)
  - Problem with compiler optimizations

<table>
<thead>
<tr>
<th>d:= b+c;</th>
<th>MOVE @B,D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD @C,D0</td>
<td></td>
</tr>
<tr>
<td>MOVE DO,@D</td>
<td></td>
</tr>
</tbody>
</table>

| d:= d+a; | ADD DO,@A |

- The variations may be taken into account (parameterized schemas) ex:

  \( T(S) = T(\text{var}, \text{var\_type}) + T(=, \text{var\_type}, \text{exp\_type}) + T(\text{exp}, \text{exp\_type}) \)

- Long blocks
- One entry and one exit point
- Example: a few consecutive assignment statements

- Code prediction
  - Same algorithm than the compiler
  - From the generated code

- Very good match with measurement
**Time schemas - results**

<table>
<thead>
<tr>
<th>Construction</th>
<th>Prediction (short blocks)</th>
<th>Prediction (long blocks)</th>
<th>Measured time</th>
<th>Corrected measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Expressions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a+b</td>
<td>[1.42, 3.46]</td>
<td>[3.46, 3.46]</td>
<td>3.66</td>
<td>3.46</td>
</tr>
<tr>
<td>a*b</td>
<td>[5.29, 7.32]</td>
<td>[6.31, 6.31]</td>
<td>6.43</td>
<td>6.31</td>
</tr>
<tr>
<td>a&gt;b</td>
<td>[2.44, 4.07]</td>
<td>[3.86, 3.86]</td>
<td>3.97</td>
<td>3.86</td>
</tr>
<tr>
<td><strong>Assignments</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a=b</td>
<td>[1.22, 2.03]</td>
<td>[2.03, 2.03]</td>
<td>2.15</td>
<td>2.03</td>
</tr>
<tr>
<td>a=b+c</td>
<td>[1.63, 4.07]</td>
<td>[3.66, 3.66]</td>
<td>3.91</td>
<td>3.66</td>
</tr>
<tr>
<td>IF simple</td>
<td>[3.46, 4.48]</td>
<td>[3.86, 4.48]</td>
<td>[3.98, 4.71]</td>
<td>[3.86, 4.48]</td>
</tr>
<tr>
<td>WHILE simple</td>
<td>[1.83, 6.71]</td>
<td>[2.24, 6.31]</td>
<td>[2.34, 6.62]</td>
<td>[2.24, 6.31]</td>
</tr>
<tr>
<td>Empty procedure call</td>
<td>[6.72, 26.44]</td>
<td>[6.72, 6.72]</td>
<td>6.97</td>
<td>6.72</td>
</tr>
</tbody>
</table>

**Time schemas – results (2)**

- For non trivial programs, the long block approach is the only one practicable.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Long block calculation</th>
<th>Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td>[187.17, 3450.10]</td>
<td>[192.2, 2071.1]</td>
</tr>
<tr>
<td>Scheduler</td>
<td>[51.88, 4253.31]</td>
<td>[280.68, 3242.6]</td>
</tr>
<tr>
<td>search process list</td>
<td>[25.84, 838.21]</td>
<td>[37.14, 826.6]</td>
</tr>
<tr>
<td>allocate idle</td>
<td>[25.02, 129.29]</td>
<td>[52.26, 127.2]</td>
</tr>
<tr>
<td>preempt</td>
<td>[105.38, 225.02]</td>
<td>[113.72, 230.7]</td>
</tr>
</tbody>
</table>

**Time schemas – analysis**

- Close to table lookup
- Control structures may be taken into account
- Gives maximum as well as minimum execution time
- It is also possible to take into account dynamic memory refreshing and periodic interrupts (from timers)
- Approach limited to processors without cache and without pipeline

**Extended Time schemas**

- Take cache and pipeline into account
- Only for worst case execution time
- Calculation gives a higher result than measurement
- The current version does not eliminate the paths that are not used at execution time
Extended Time schemas

- They exhibit the advantages of the basic time schemas
  - But do not give the (min.) best execution time
- Take into account the effects of pipeline
- Take into account the effects of caches (data and program)
- Not simple
- Give results that are still much higher than measurements
  - Limits of the current models

Conclusion

- Two simple techniques
  - Measurements
  - Instruction counting
- With these techniques, there is no guarantee to find the worst case execution time
  - You should always include overload protection in your scheduling
- For simple processors (microcontrollers), calculating the execution time (min and max) is feasible
  - Better than measurement in terms of risk of exceeding the highest value
- For complex processors, new techniques are coming that are (much) more complicated with results that still need to improve