IN4343 Real-Time Systems

Practical Timing Analysis
Dynamic Timing Analysis

Definition

- Full study of a program to determine its worst case execution time
  - it is based on program execution(s)
Dynamic Timing Analysis

Characteristics

• Result is specific to
  - underlying hardware platform
  - memory management
  - application context / input data

• Accuracy depends on coverage, but always underestimates true WCET => unsafe

• Two basic approaches
  - Instruction tracing
  - Execution timing
Dynamic vs. Static TA

- Dynamic Timing Analysis
  - Underestimate
  - Poorer Estimates

- Static Timing Analysis
  - Overestimate
  - Actual
  - Worst-Case Execution Time
  - Effort Timing Analysis

- Poorer Estimates

TU Delft
embedded software
Dynamic vs. Static TA

- Poorer Estimates
- Actual Worst-Case Execution Time
- Static Timing Analysis
- Overestimate
- Underestimate
- Dynamic Timing Analysis
- Effort Timing Analysis

This *cannot* happen under any circumstance

This can happen, if poorly performed
Dynamic Timing Analysis

Problems

• Measuring all possible paths is infeasible
  ➢ combinatorial explosion
  ➢ $10^{40}$ paths for mid-size task

• Partitioning does not work
  ➢ combining WCET of parts may not yield WCET of whole

• Requires input scenarios
  ➢ high coverage required
  ➢ rare paths (exceptions) may be missed
Dynamic Timing Analysis

Still useful?!

- Soft real-time systems
  - WCET may be off (a little)

- Easy to use
  - different hardware platforms
  - quick rough estimate

- Aid static analysis
  - “hard” evidence
  - lower bound
Industrial Approach

Example of Industrial Design Flow

Design
Matlab + Simulink
Matlab + RTW

Test
Prototype Boards
Custom Hardware

Deploy
Custom automotive Hardware

[Peter Puschner et al.]
Industrial Approach
Industrial Approach – Input Vectors
Industrial Approach – Random Data

[Peter Puschner et al.]
Industrial Approach – Pitfalls

- Test-coverage metrics for functional tests are not sufficient for a measurement-based WCET assessment
- Random data may miss the path with the longest Execution Time
- The state of the system is typically not taken into consideration
Dynamic Timing Analysis

Pragmatic approach

• What you don’t execute you cannot measure!
• What you don’t cover you cannot execute!
• Try to cover as much as possible!
  ➢ may skip obviously shorter branches

• Leverage software testing framework
  ➢ code coverage for free
  ➢ additional test cases for
    • max number of loops
    • max recursion depth
    • …
Basic Coverage Criteria

- Function coverage
- Statement coverage
- Branch coverage
- Condition coverage
  - exercise all sub conditions to T/F
  - does not imply branch coverage

- Modified condition/decision coverage
  - branch + condition coverage
  - sub condition decides outcome

int foo (int x, int y) {
    int z = 0;
    if (x>0 && y>0) {
        z = x;
    }
    return z;
}
Software Testing Fundamentals

• Program is always executed as a black box (otherwise it is called debugging)

• Structural information (white-box information) is only used to derive test cases

• The tester needs to identify input situations
  ➢ that lead to the execution of all entities
  ➢ according to the predefined coverage measures

• A coverage tool is quite useful
Coverage tools

**gcov**

```bash
$ gcc -fprofile-arcs -ftest-coverage tmp.c
$ a.out
Success
```

```c
#include <stdio.h>

int i, total;

total = 0;

for (i = 0; i < 10; i++)
    total += i;

if (total != 45)
    printf("Failure\n");
else
    printf("Success\n");
```

```bash
$ gcov tmp.c
87.50% of 8 source lines executed in file ‘tmp.c’
Creating ‘tmp.c.gcov’
```
Coverage tools
www.ECLEmma.org

- Eclipse plugin for Java
Avoiding Execution Time Analysis
A practical alternative

- **Static TA**
  - loose upper bound

- **Dynamic TA**
  - lower bound => unsafe

- Write temporally predictable code
  - Single-path code

↓ Predicated execution

```
if cond
  res := expr1
  res := expr2
```

```
P := cond
(P) res := expr1
(not P) res := expr2
```
Time-Predictable Single-Path Code

Don’t let the environment dictate

- Sequence of actions
- Durations of actions

Take control decisions offline!!

Single-path code:

- no input-data dependent branches
- predicated execution (poss. with speculation)
- control-flow orientation → data flow focus

[Peter Puschner et al.]
Branching vs. Predicated Code

Code example: \( \text{if } rA < rB \text{ then } \text{swap}(rA, rB); \)

Branching code
\[
\begin{align*}
\text{cmplt } rA, rB \\
\text{bf } & \text{skip} \\
\text{swp } & \text{rA, rB} \\
\text{skip: }
\end{align*}
\]

Predicated code
\[
\begin{align*}
\text{predlt } Pi, rA, rB \\
(Pi) & \text{ swp } \text{rA, rB}
\end{align*}
\]
HW-Support for Predicated Execution

Predicate registers

Instructions for manipulating predicates
  (define, set, clear, load, store)

Predicated instructions
  • Support for full predication: execution of all instructions is controlled by a predicates
  • Support for partial predication: limited set of predicated instructions
    (e.g., conditional move, select, set, clear)
Example: Speedup by if-conversion

\[
\text{if } rA < rB \text{ then } \text{swap}(rA, rB);
\]

**Branching code**
- cmplt rA, rB
- bf skip
- swp rA, rB
- skip:

**Predicated code**
- predlt Pi, rA, rB
- (Pi) swp rA, rB

Execution in three-stage pipeline:
- IF: 5 cycles
- DE: 6 cycles
- EX: 4 cycles

[Peter Puschner et al.]
Implications of Partial Predication

Speculative code execution

- instructions that do not allow for predication are executed unconditionally, and
- the results are stored in temporary variables;
- subsequently, predicates determine which values of temporary variables are further used

Example:

```
(pred)       cmov dest, src1
(not pred)   cmov dest, src2
```

Cave: speculative instructions must not raise exceptions!
(e.g., div. by zero, referencing an invalid memory address)
Fully vs. Partially Predicated Code

Original code:

\[
\text{if } \text{src2} \neq 0 \text{ then } \text{dest} := \text{src1} / \text{src2};
\]

Fully predicated code:

\[
\text{Pred} := (\text{src2} \neq 0)
\]

\[
(\text{Pred}) \div \text{dest, src1, src2}
\]
Fully vs. Partially Predicated Code (2)

Original code:

```plaintext
if src2 \neq 0 \textbf{then} \text{dest} := \text{src1} / \text{src2};
```

Partially predicated code, first attempt:

```plaintext
\text{Pred} := (\text{src2} \neq 0)
\text{div } \text{tmp\_dst, src1, src2}
\text{(Pred) cmov } \text{dest, tmp\_dst}
```

may raise an exception on division by zero
Fully vs. Partially Predicated Code (3)

Original code:

```plaintext
if src2 ≠ 0 then dest := src1 / src2;
```

Partially predicated code:

```plaintext
Pred := (src2 ≠ 0)
```

<table>
<thead>
<tr>
<th>(not Pred)</th>
<th>mov</th>
<th>tmp_src, src2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cmov</td>
<td>tmp_src, $safe_val</td>
</tr>
<tr>
<td></td>
<td>div</td>
<td>tmp_dst, src1, tmp_src</td>
</tr>
</tbody>
</table>

| (Pred)     | cmov  | dest, tmp_dst |
```
“Minimal” Predicated-Exec. Support

Conditional Move instruction:

\texttt{movCC \textit{destination}, \textit{source}}

Semantics:

\begin{verbatim}
    if \textit{CC} \\
        then \textit{destination} := \textit{source} \\
    else no operation
\end{verbatim}
If-conversion with conditional move

\[
\begin{align*}
&\text{if } \text{cond} \\
&\text{res} := \text{expr1} \\
&\text{res} := \text{expr2} \\
&\text{t1} := \text{expr1}' \\
&\text{t2} := \text{expr2}' \\
&\text{test } \text{cond} \\
&\text{movT res, t1} \\
&\text{movF res, t2}
\end{align*}
\]

\{ avoid side effects! \}
Emulation of conditional move

In architectures without predicate support, conditional moves can be emulated with bit-mask operations.

Example:

```plaintext
if (cond) x=y; else x=z;
```

- \( t0 = 0 – \text{cond}; \)  // fat bool: 0..false, -1..true
- \( t1 = \neg t0; \)  // bitwise negation (fat bool)
- \( t2 = t0 \& y; \)
- \( t3 = t1 \& z; \)
- \( x = t2 | t3; \)

assumption: the types of all values have the same size
Example

for (i = SIZE-1; i > 0; i--) {
    for (j = 1; j <= i; j++) {
        if (a[j-1] > a[j]) {
            t = a[j];
            a[j] = a[j-1];
            a[j-1] = t;
        }
    }
}

Bubble sort: input array a[SIZE]

for (i = SIZE-1; i > 0; i--) {
    for (j = 1; j <= i; j++) {
        t1 = a[j-1];
        t2 = a[j];

        if (t1 > t2) {
            t = a[j];
            a[j] = a[j-1];
            a[j-1] = t;
        }
    }
}
Example

```
for(i=SIZE-1; i>0; i--)
{
    for(j=1; j<=i; j++)
    {
        if (a[j-1] > a[j])
        {
            t = a[j];
            a[j] = a[j-1];
            a[j-1] = t;
        }
    }
}
```

Bubble sort: input array a[SIZE]

```
for(i=SIZE-1; i>0; i--)
{
    for(j=1; j<=i; j++)
    {
        cond = (a[j-1] > a[j]);
        (cond): t = a[j];
        (cond): a[j] = a[j-1];
        (cond): a[j-1] = t;
    }
}
```
Example

for (i = SIZE-1; i > 0; i--)
{
    for (j = 1; j <= i; j++)
    {
        if (a[j-1] > a[j])
        {
            t = a[j];
            a[j] = a[j-1];
            a[j-1] = t;
        }
    }
}

Bubble sort: input array a[SIZE]

for (i = SIZE-1; i > 0; i--)
{
    for (j = 1; j <= i; j++)
    {
        t1 = a[j-1];
        t2 = a[j];
        test (t1 > t2);
        movt: a[j-1] = t2;
        movt: a[j] = t1;
    }
}
Single-Path Properties

Every execution has the same instruction trace, i.e., the same sequence of references to instruction memory.

Path analysis is trivial – there is only one path.

Two executions starting from the same instruction-cache state have identical hit/miss sequences on accesses to instruction memory.
Example

for(i=SIZE-1; i>0; i--)
{
    for(j=1; j<=i; j++)
    {
        if (a[j-1] > a[j])
        {
            t = a[j];
            a[j] = a[j-1];
            a[j-1] = t;
        }
    }
}

Bubble sort: input array a[SIZE]

for(i=SIZE-1; i>0; i--)
{
    for(j=1; j<=i; j++)
    {
        t1 = a[j-1]; t2 = a[j];
        test (t1 > t2);
        movt: t = t2;
        movt: t2 = t1;
        movt: t1 = t;
        a[j-1] = t1; a[j] = t2;
    }
}

identical data access pattern!

[Peter Puschner et al.]
What about loops?

while (cond) {
    body;
}

int done = 0;
for (int i = 0; i<MAX_LB; i++) {
    if (!done && cond) {
        body;
    } else {
        done = 1;
    }
}
Transformation Properties

Completeness: every piece of code with boundable WCET can be transformed

Transformed code has a single path

WCET analysis is trivial: execute and measure

WCET analysis yields exact WCET

Execution times are long (if we are not careful)
Performance of Single-Path Code

Execution times of input-dependent alternatives sum up due to serialization

⇒ Execution times of single-path code are long if the control flow of its source is strongly input dependent
Performance of Single-Path Code (2)

CPUs with deep pipelines need a number of cycles to re-fill the pipeline after a (mis-predicted) branch

⇒ predicated execution can be cheaper than jumping

⇒ this is where modern compilers/processors use predicated execution to improve performance
Execution Times

Code execution times before and after single-path transformation

[Peter Puschner et al.]
Practical Execution Time Analysis

Summary

• Dynamic TA
  ➢ used in industry
  ➢ quick and easy
  ➢ underestimates WCET => soft real time

• Single-path transformation
  ➢ predicated code
  ➢ automatic conversion [see reading list]
  ➢ execution time goes up (a lot)