Compilation and Worst-Case Execution-Time Analysis

Peter Puschner
Why Compiler-Support for WCET Analysis?

Compiler “knows” both code representations

- Source code (the user representation)
- Object code (the instructions determining timing)

Compiler “knows” the transformations it applies

- optimizations could target the worst-case path

谖 Compiler can generate information that is relevant for WCET analysis

谖 Compiler can generate time-predictable code (cache locking, pre-fetching for scratch-pad, single-path code, avoidance of timing anomalies, etc.)
A Related Problem – Source-level debugging

Source-level debugging faces similar challenges as timing analysis of optimized code:

- Mapping between source code and object code becomes blurred due to code transformations.
- In debugging, code locations and store (memory, variables, etc.) content have to be mapped.
- In timing analysis, flow information has to be mapped to object code and timing results need to be mapped to the source code.
A Related Problem – Source-level debugging

int test (int a)
{
    int b = a;
    int c = b + 4;
    return c;
}

What is the value of variable 'b'??

→ additional information required to reconstruct original variable values.

int test (int a)
{
    int c = a + 4;
    return c;
}

optimization
Contents

Provide program information to reduce manual code annotations
WCET-aware code optimizations
Transformation of flow information
Preservation of code coverage
Reducing Manual Code Annotations

Why is it hard to analyse code semantics for WCET analysis?

- Platform-specific behavior (memory layout, ROM-areas, memory-mapped IO, representation of data types, …)
- Control-flow is well hidden within low-level constructs of object code (indirect jumps, …)
- Formal limits of computability (loop bounds, correlation of code predicates, type-inference in object-oriented languages, …)
Workaround: Guide the Analysis by Means of Code Annotations (1)

Annotations to find jump-targets [aiT]:

- instruction <addr> calls <target-list>;
- instruction <addr> branches to <target-list>;

Labeling assembly-instructions with high-level meaning [aiT]:

- instruction <addr> is a return;
Workaround: Guide the Analysis by Means of Code Annotations (2)

Describing possible values of variables [aiT]:

- condition <addr> is always <bool value>;
- snippet <addr> is never executed;
- instruction <addr> is entered with <state>;

Describe boundaries of memory access [aiT]:

- instruction <addr> accesses <addr-range>;
Workaround: Guide the Analysis by Means of Code Annotations (3)

Describe iteration bounds of loops [Bound-T]:

subprogram “<fn-name>”
  loop that uses <var-name>
    repeats <= 7 times;
  end loop;
end “<fn-name>”;

(information in external code annotation file)
Reflection on Code Annotations

Code annotations may be erroneous

Often code has to be annotated at object code level ⇒
update/check annotations after changes and re-compilation
of code

Manually annotating the code is labor-intensive (especially
at the object-code level !!!)

Questions:

Are there code annotations that can be avoided?
Can we simplify the annotation process?
Classification of Code Annotations (1)

Platform Property Annotations (PPA)
CFG Reconstruction Annotations (CRA)
Program Semantics Annotations (PSA)
Auxiliary Annotations (AA)
Classification of Code Annotations (2)

Platform Property Annotations (PPA)

Description of special semantics behind the access of memory-mapped IO (local annotations).

Description of platform properties that influence the execution time (mostly global annotations).

Examples:
  description of memory layout, properties of target hardware, …
Classification of Code Annotations (3)

CFG Reconstruction Annotations (CRA)

Help to build basic structures of code analysis:

• control-flow graph (CFG) and call-graph

Allow to reverse-engineer the high-level code structure from the low-level assembly statements.

Examples:
list of targets for branch instructions, type labeling of branch instructions, …
Classification of Code Annotations (4)

Program Semantics Annotations (PSA)
Local description of program behavior
Support for
  • path analysis
  • value analysis
  • etc.
Examples:
  flow annotations like loop bounds, description of variable values, …
Compiler-Support for WCET Analysis (1)

Information accessible by the compiler:
Program representation at source and object code level.
Code transformations performed during compilation.

How the compiler can help:
Describe **CFG of object code** (to avoid the need of CRA)
Provide mapping of source-code annotations to object code (simplification by writing annotations at source code level instead of object-code level) [PPA, PSA, AA]
Compiler-Support for WCET Analysis (2)

How the compiler can help (2):

Emit information about a program’s execution behavior (reduce amount of code annotations [PSA])
  • in general, some PSA may remain mandatory
  • but in practice almost no PSA is mandatory!

Improve predictability of code
  • imitation of “WCET-oriented programming”
  • e.g., single-path conversion can reduce the execution-time jitter of real-time programs
  • support of predictable HW mechanisms (e.g., prefetching, scratchpad memory, cache locking),
Final Remark on Code Annotations

Not all code annotations are artificial:
application-context (operation mode) may be specified to refine the analysis [PSA] (e.g. range on initial variable values, CFG restriction)

mapping of target-specific properties to code may be mandatory [PPA] (e.g. memory-mapped IO)
Contents

Provide program information to reduce manual code annotations

WCET-aware code optimizations

Transformation of flow information

Preservation of code coverage
WCET-aware Code Optimizations

- localize worst-case path,
- select appropriate optimizations,
- identify code locations to apply optimizations
Invariant Code Motion – a real-time optimization?

what if loop is never executed? \(\rightarrow\) WCET is increased

if loop is executed, optimized code has decreased WCET
WCET-aware Code Optimizations (3)

Superblock Formation:

superblock: …block with single entry point

increase of code size but reduction of WCET
Worst-Case Path

Be aware: optimizations may change the worst-case path
Worst-Case Path

Be aware: execution of new worst-case path may take longer than the original one
Contents

Provide program information to reduce manual code annotations

WCET-aware code optimizations

Transformation of flow information

Preservation of code coverage
Challenge: Keeping Flow Information Correct

Q: How can we guarantee that flow facts for compiled code are correct?

parallel transformation of flow facts
Challenge: Keeping Flow Information Correct (2)

for (i=0; i<N; i++)
maximum 10 iterations
{
    f(i);
}

for (i=0; i<(N-2); i=i+3)
{
    f(i); f(i+1); f(i+2);
}

for (;i<N; i++)
{
    f(i);
}

additional knowledge assumed: N ≤ 10
(flow fact given as code annotation)

Q: what flow facts are known of the transformed code?
Challenge: Keeping Flow Information Correct (3)

for (i=0; i<N; i++)
maximum 10 iterations
{
    f(i);
}

Exercise: explain why the new flow facts follow from the original code and the transformation

additional knowledge assumed: N ≤ 10 (flow fact given as code annotation)
Challenge: Keeping Flow Information Correct (4)

Flow information (flow facts) to be transformed:

- **Loop bound information** (upper and lower bound of loop iterations)
- **Marker bindings** (link between flow variables and control-flow edges of the program), e.g., “mAB” denotes the control flow from CFG node A to node B.
- **Flow constraints** (linear relations between flow variables), e.g., “4*mAB ≤ 1*mCD”
Flow-Facts Transformation Functions $F_{ff}()$

A flow facts transformation function $F_{ff}()$ has to be derived from the given code structure and the applied code optimization/ transformation function $F_{code}()$.

$F_{code}()$ is composed of a sequence of low-level operations, like jump redirection; move, duplication, deletion, modification of code, etc.

Deriving $F_{ff}()$ directly from the low-level operations of $F_{code}()$ is not possible, since low-level operations may leave the program in an inconsistent form – only after all low-level operations has the program to be semantically equivalent to the original program.

$\Rightarrow F_{ff}()$ has to be derived from the $F_{code}()$ as a whole.
Flow-Facts Transformation Functions $F_{ff}()$

Flow facts transformation functions are composed of the following operations:

- **Update of loop bound information** ($L$): (create, modify, or delete loop bound information)
- **Update of flow constraints** ($R$): (transform terms of the form “$\text{const} \cdot \text{flowvariable}$” of a linear flow constraint into a new term respectively a sum of terms, creation of new flow constraints)
Results of Flow Facts Transformation

- Example: bubble sort
  - algorithm: a pair of nested loops where the iteration count of the inner loop depends on the current iteration of the outer loop (\(\rightarrow\) “triangular loop”)
- Compiler: gcc 2.7.2 (C167 port by HighTec), extended to handle flow information (TU Vienna)
- Static WCET tool: calc_wcet_167 (TU Vienna)
# Results of Flow Facts Transformation

<table>
<thead>
<tr>
<th>WCET analysis method</th>
<th>nonoptimized code [cycles]</th>
<th>optimized code [cycles]</th>
<th>improvement [rel.]</th>
</tr>
</thead>
<tbody>
<tr>
<td>measured</td>
<td>104.007</td>
<td>30.486</td>
<td>70.7%</td>
</tr>
<tr>
<td>static 1 (loop bounds only)</td>
<td>198.741</td>
<td>57.816</td>
<td>70.9%</td>
</tr>
<tr>
<td>static 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

→ compatibility of WCET analysis framework with code optimizations provides significant WCET improvement
## Results of Flow Facts Transformation

<table>
<thead>
<tr>
<th>WCET analysis method</th>
<th>nonoptimized code [cycles]</th>
<th>nonoptimized code [rel.]</th>
<th>optimized code [cycles]</th>
<th>optimized code [rel.]</th>
</tr>
</thead>
<tbody>
<tr>
<td>measured</td>
<td>104.007</td>
<td>100.0%</td>
<td>30.486</td>
<td>100.0%</td>
</tr>
<tr>
<td>static 1</td>
<td>198.741</td>
<td>191.1%</td>
<td>57.816</td>
<td>189.6%</td>
</tr>
<tr>
<td>(loop bounds only)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>static 2</td>
<td>104.625</td>
<td>100.6%</td>
<td>30.492</td>
<td>100.1%</td>
</tr>
<tr>
<td>(detailed flow facts)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

→ support of detailed flow facts is important to avoid high overestimation of WCET
Contents

Provide program information to reduce manual code annotations

WCET-aware code optimizations

Transformation of flow information

Preservation of code coverage
Preservation of Code Coverage

**Assumption**: Measurement-based WCET analysis that uses automatic test-data generation based on source code analysis

- highly retargetable (source code semantics almost platform independent)

**Test data generation method**: e.g., partition the source code into smaller units (so-called segments) and generate test data to cover each path within a segment
Preservation of Code Coverage (2)

Typical structural code coverage metrics:
- statement coverage (not line coverage!)
- decision coverage (=branch coverage)
- condition coverage
- MC/DC coverage
- path coverage

Challenge: How can we assure that the code coverage at source code is also valid at object code level?
- compiler may introduce conditional control flow to optimize the frequent case
Preservation of Code Coverage (3)

Solution:
introduce *compilation profiles* into the compiler to select only optimizations that do preserve a given structural code coverage

Command line switches used to specify which level of structural code coverage should be preserved:

```
cp –sccp/dc   … preserve decision coverage
```
Summary

Compiler-support is very beneficial for WCET analysis:

- reduce amount of code annotations to be specified manually
- optimizations can be chosen to optimize the worst-case path
- generate time-predictable code (next lectures)

Question: when is the hard real-time market strong enough to demand such solutions?
Bibliography


http://ti.tuwien.ac.at/rts/teaching/courses/wcet

http://www.wcet.at