Checking Timing Behavior and Memory Usage by Abstract Interpretation of Executable Code

Dr.-Ing. Christian Ferdinand
Dr. Reinhold Heckmann
AbsInt Angewandte Informatik GmbH

Prof. Dr. Reinhard Wilhelm
Universität des Saarlandes
Hard Real-Time Systems

- Controllers in planes, cars, plants, … are expected to finish their tasks within reliable time bounds.
- Schedulability analysis must be performed
- Hence, it is essential that an upper bound on the execution times of all tasks is known
- Commonly called the Worst-Case Execution Time (WCET)
The Timing Problem

- Best Case Execution Time
- Exact Worst Case Execution Time
- Safe Worst Case Execution Time Estimate
- Unsafe: Execution Time Measurement

Execution Time

Probability
Automotive Embedded Control Software

- The Software tends to be large and complex
  - Much functionality from different providers
  - Code generator tools
    - RTOS
    - communication libraries
The Timing Problem

\[ x = a + b; \]

LOAD \ r2, \ _a  
LOAD \ r1, \ _b  
ADD \ r3, r2, r1

68K (1990)

MPC 5xx (2000)

PPC 755 (2001)

Execution Time (Clock Cycles)

Execution Time depending on Flash Memory (Clock Cycles)

Best Case  Worst Case

Execution Time (Clock Cycles)

Best Case  Worst Case
Modern Hardware vs. Predictability

- Multiple memories, caches, pipelines, branch prediction, …

- Performance depends on execution history. This makes the prediction difficult

- Software monitoring, dual loop benchmark, direct measurement with logic analyzer, hardware simulation are no longer generally applicable.

- No information means: assume the worst

- Switching off caching reduces performance by a factor of 30 (EADS study)
Some Architectural Features that make Measurement-Based WCET Analysis a Challenge

- The empty cache is not necessarily the “worst case cache”
- “Domino” effects
- The global round robin counter/PLRU state bits can be changed by interrupt routines
- A cache miss is not necessarily the worst case
Solution: Static WCET Analysis

- The WCET analyzer computes safe upper bounds of the execution times of the tasks in a program for all inputs.

- Static program analysis based on Abstract Interpretation.

- The analysis design is proven to be correct.
**aIT WCET Analyzer**

A Solution to the Timing Problem

- Input: an executable program, starting points, loop iteration counts, call targets of indirect function calls, and a description of bus and memory speeds
- Computes **Worst-Case Execution Time** bounds of tasks
aiT WCET Analyzer Structure

Executable program

CFG Builder

Loop Trafo

CRL File

Static Analyses
- Value Analyzer
- Cache/Pipeline Analyzer

AIP File

PER File

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

Loop bounds

WCET Visualization

AIP File

CRL File
# StackAnalyzer Results

## Maximum System Stack Usage

<table>
<thead>
<tr>
<th>Task</th>
<th>Stack Usage</th>
<th>Sum of</th>
<th>Max of</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicTaskFirst_system</td>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallGraph basicTaskFirst_func</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>max StackOffset</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StaticOffset</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>basicTaskFirst_user</td>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CallGraph basicTaskFirst_func</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>max StackOffset</td>
<td>8</td>
<td>max</td>
<td></td>
</tr>
<tr>
<td>StaticOffset</td>
<td>32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Category 1 ISR

- **Task context offset**

## Maximum User Stack Usage

(osCAN C16x/ST10)
Pipeline Analysis
aiT: Timing Details

Worst Case Execution Time: 4835

routine: _prime

Min. Iteration: 0
Max. Iteration: 22
Predicted WCET Contribution: TBD
Context 0: 1 loop (0000000000000000)
Context 1: 1 loop (0000000000000000)

routine: _even

routine: U_MOD

AbsInt
Angewandte Informatik
**aiT: Timing Details**

```c
int n { divides (2, n) ;
    #: 1
    t: 12
} ;

routine: loop_0001

if (divides (i, n) ) / ai: loop here min 0 max 357

max #: 357
max t: 9

<empty>

max #: 357
max t: 4

max #: 357
max t: 2
```
aiT: Timing Details
Interprocedural Analysis/Analysis of Loops

- Loops are analyzed like procedures
- This allows for:
  - Virtual inlining
  - Virtual unrolling
  - Better address resolution
  - Burst accesses
  - Selectable precision
- Optional user constraints

```
routine: routine_MUCBGIDB

routine: routine_36S036D6
```

```
Context 0: count=1, time=2
Context 1: count=1, time=3
Context 2: count=1, time=4
Context 3: count=1, time=4
Context 4: count=1, time=4
Context 5: count=1, time=4
Context 6: count=1, time=4
Context 7: count=1, time=4
Context 8: count=1, time=4
Context 9: count=1, time=4
Context 10: count=1, time=4
Context 11: count=1, time=4
Context 12: count=1, time=4
Context 13: count=1, time=4
Context 14: count=1, time=4
Context 15: count=1, time=4
Context 16: count=1, time=4
```
Challenge: Reconstruction of CFG

- Indirect Jumps
  - Case/Switch statements as compiled by the C-compiler are automatically recognized
  - For hand-written assembly code annotations might be necessary

```
INSTRUCTION ProgramPoint BRANCHES TO Target_1, ..., Target_n
```

- Indirect Calls
  - Can often be recognized automatically if a static array of function pointers is used
  - For other cases

```
INSTRUCTION ProgramPoint CALLS Target_1, ..., Target_n
```
Loops

- aiT includes a loop bound analysis based on interval analysis and pattern matching that is able to recognize the iteration count of many „simple“ FOR loops automatically.

- Other loops need to be annotated
  
  Example:
  ```
  loop "\_prime" + 1 loop end max 10;
  ```
Source Level Annotations

```c
bool divides (uint n, uint m) {
    /* ai: SNIPPET HERE NOT ANALYZED, TAKES MAX 173 CYCLES; */
    return (m % n == 0);
}

bool prime (uint n) {
    uint i;
    if (even (n))
        /* ai: SNIPPET HERE INFEASIBLE; */
        return (n == 2);
    for (i = 3; i * i <= n; i += 2) {
        /* ai: LOOP HERE MAX 20; */
        if (divides (i, n))
            return 0;
    }
    return (n > 1);
}
```
aiT WCET Analyzer Advantages

- aiT WCET analyzer allows you to:
  - inspect the timing behavior of (timing critical parts of) your code

- The analysis results
  - are determined without the need to change the code
  - hold for all inputs and all executions (for the intrinsic cache and pipeline behavior)
Preliminary feedback from automotive users

- Providing the annotations (targets of indirect function calls and loop bounds) can require some effort
- Precision ✓
  - “Arbitrary ILP constraints” basically not used
- Analysis speed ✓
- Integration into development process still to be done
Conclusion

- **aiT** enables development of complex hard real-time systems on state-of-the-art hardware.

- Increases safety.

- Saves development time.

- Precise timing predictions enable the most cost-efficient hardware to be chosen.
Future Work at Saarland University

- „Compositional“ WCET analysis