Removing Infeasible Paths in WCET Estimation: The Counter Method

Work made during the ANR Project W-SEPT (2012-2016)

Mihail Asavoae, Rémy Boutonnet, Fabienne Carrier, Nicolas Halbwachs, Erwan Jahier, Claire Maiza, Catherine Parent-Vigouroux, Pascal Raymond

Verimag/Grenoble-Alpes University

SYNCHRON16, dec. 2016, Bamberg
A brief introduction on WCET and IPET

WCET estimation

- Dynamic methods (test) give realistic, feasible exec. times, but are not safe
- Static methods (WCET analysis) give guaranteed upper bound to exec. time, but necessarily overestimated
- Main sources of over-approximation:
  - Hardware (too complex, abstractions)
  - Software (infeasible paths)
WCET tool organization

- Value analysis:
  ↦ gives info on the program semantics
  ↦ in particular *loop bounds*

- Control Flow Graph (CFG) construction:
  ↦ Basic Blocks (BB) of sequential instructions
  ↦ connected by transitions (jump/sequence)

- Micro-architecture analysis:
  ↦ assigns local WCET to each BB/transitions
  ↦ according to a more or less precise model
  ↦ N.B. given in cpu cycles

- Find the worst path in the CFG
  ↦ widely used method: IPET
    (Implicit Path Enumeration Technique)
  ↦ based on Integer Linear Programming encoding (ILP)
IPET on an example

- $\mu$-archi analysis has assigned weights
e.g. $w_a = 26$, $w_b = 72$ etc.
- data-flow analysis has found loop bounds
'\(h\)' taken at most $n = 10$ times
- ILP encoding:
  - Structural constraints
    \[ a + d = g = p = 1 \]
    \[ g + k = p + h \]
    \[ h = e + b = f + c = k \]
  - Semantic constraints
    \[ h \leq n = 10 \]
  - Objective: $\text{MAX}\left( \sum_{x \in \mathcal{E}} w_x x \right)$
  - Solution: $a=g=p=1, h=b=c=k=10, d=e=f=0$
    with: $26+7+7+10*(5+72+68+5) = 1540$
- Extra semantic info: $b$ and $c$ exclusive at each iteration
  - Can be expressed with $b+c \leq n = 10$
  - Solution: $a=g=p=1, h=e=c=k=10, d=b=f=0$
    with: $26+7+7+10*(5+50+68+5) = 1320$
Semantic properties and WCET estimation

Idea/goal

- use *state of the art static analysers* to enhance *state of the art WCET estimation* ...

- ... implies some choices:
  - program analysis at the C level (that’s what program analyzers do...)
  - comply the IPET/ILP approach (that’s what WCET analyzers do...)

How/technique

Briefly, instrument the program with *control-flow points counters*:

- Static C program analyzers are likely to discover invariants relations between integer variables (e.g. linear static analysis à la Halbwachs/Cousot)
- This kind of relations perfectly meet the IPET/ILP approach
Static analysis to linear constraint: example

From principles to practice...

- Which C program to consider?
- How to relate (C) counters with (binary) basic blocks?
- Integration in the WCET work-flow?
Tools/Technical choices

- **OTAWA+lp_solve** for WCET/IPET and ILP
- **pagai**, (Henry/Monnaiaux/Boutonnet) for linear analysis
- **Cil/Frontc library** for C program manipulation
- **arm-elf-gcc**
- Case studies: Tacle Bench + some others (Lustre/Scade)

Note on loop bounds

- We know that linear analysis is NOT a good method for finding (nested) loop bounds
- We generally use **ORANGE** (from OTAWA lib) to find loop bounds
Work-flow “meta” steps

- bounds pragmas
- Ref. C code
- Frontend (instrumentation)
  - bounds checking (orange and/or pragmas)
  - ref. ilp system
  - Backend (owcet, pagai, pagai2lp) (lp_solve)
    - Ref. C code
    - Ref. C code + counters
    - Ref. bin code
    - counters 2 BBs info

original C code
- 2 estimations + logs
- Ref. C code + counters
- Ref. bin code
- counters 2 BBs info

Semantic properties and WCET estimation
To do

- Add counters (at least !)
- ... but also get rid of unsupported constructs (owcet and/or pagai)
  - preprocessing directives,
  - multiple returns,
  - computed gotos, switches ...
  - ... and plenty of NL's (to help line-by-line traceability) !
- and keep trace of user annotations (if any, e.g. bounds pragma)

- Notion of reference program:
  - free of undesired features
  - semantically equivalent
  - structurally, as close as possible
  - same reference for program analysis and timing analysis (via compilation)
Running example: lcdnum.c (from Mälardalen)

```c
#ifndef PROFILING
#include <stdio.h>
#endif

unsigned char num_to_lcd(unsigned char a) {
    switch(a) {
    case 0x00: return 0;
    case 0x01: return 0x24;
    case 0x02: return 1+4+8+16+64;
    case 0x03: return 1+4+8+32+64;
    case 0x04: return 2+4+8+32;
    case 0x05: return 1+4+8+16+64;
    case 0x06: return 1+2+8+16+32+64;
    case 0x07: return 1+4+32;
    case 0x08: return 0x7F;
    case 0x09: return 0x0F + 32 + 64;
    case 0x0A: return 0x0F + 16 + 32;
    case 0x0B: return 2+8+16+32+64;
    case 0x0C: return 1+2+16+64;
    case 0x0D: return 4+8+16+32+64;
    case 0x0E: return 1+2+8+16+64;
    case 0x0F: return 1+2+8+16;
    }
    return 0;
}

volatile unsigned char IN = 120;
volatile unsigned char OUT;

int main(void) {
    #ifdef PROFILING
    int iters_i = 0, min_i = 100000, max_i = 0;
    #endif
    int i;
    unsigned char a;
    #ifdef PROFILING
    iters_i = 0;
    #endif
    #Pragma("loopbound min 10 max 10")
    for(i=0; i<10; i++) {
        #ifdef PROFILING
        iters_i++;
        #endif
        a = IN;
        if(i<5) {
            a = a & 0x0F;
            OUT = num_to_lcd(a);
        }
    }
    #ifdef PROFILING
    if ( iters_i < min_i ) min_i = iters_i;
    if ( iters_i > max_i ) max_i = iters_i;
    printf("i-loop: [%d, %d]\n", min_i, max_i);
    #endif
    return 0;
}
```

Frontend (Instrumentation)
Running example (cntd)

- pre-process (cpp)
- multiple returns/switch (cil)
- get a reference C program, in two versions:
  - with counters (for pagai)
  - without counters (for ORANGE and gcc then owcet)
- keep trace of:
  - counters source line
  - user-given bounds

Note: only main is shown, num_to_lcd is much bigger due to switch/return normalization.
Running example (cntd)

- Reference program is compiled: `lcd_num.elf`...
- ... and counters are associated to (binary) BB, **as far as possible:**
  → we rely on OTAWA’s `dumpcfg`, to be sure to agree on BB numbering/source line
  ← as usual, rather *fragile*, suppose that C and bin cfgs (almost) map...

*We’ll discuss later on compiler optimization*

- C line / BB mapping of the example:

<table>
<thead>
<tr>
<th>line(s)</th>
<th>bloc(s)</th>
<th>reliable</th>
<th>counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>136,144</td>
<td>1</td>
<td>yes</td>
<td><code>cptr_main.1</code></td>
</tr>
<tr>
<td>145</td>
<td>1;2</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>147,148</td>
<td>4</td>
<td>yes</td>
<td><code>cptr_main.2</code></td>
</tr>
<tr>
<td>150,151,152</td>
<td>5</td>
<td>yes</td>
<td><code>cptr_main.3</code></td>
</tr>
<tr>
<td>155</td>
<td>6</td>
<td>yes</td>
<td><code>cptr_main.4</code></td>
</tr>
<tr>
<td>158,159,160</td>
<td>3</td>
<td>yes</td>
<td><code>cptr_main.5</code></td>
</tr>
</tbody>
</table>
Instrumentation: detailed work-flow and options

options: optim
dflt -O0
maybe others (?)

original C code

cpp

options: one-return
inline
no switch

ref. C+counters

cdig -counters
(based on Frontc/CIL)

cptr2bb

counter/BB

pragma.ffx
(bound/line)

otawa’s dumpcfg

counter/line

line/BB

counter/BB

ref. BIN program

ref. C program

(gcc)

(for orange) (for owcet)

(for pagai to ilp) (for bounds seeking) (for pagai)
Bounds seeking

Sources of bounds info

- User-given bounds (e.g. Mälardalen’s *pragmas*)
- C-ref program analysis by *Orange*
- A hand-made “data-base” of standard libraries bounds, e.g.

  <loop source="gcc-4.4.2/.*/arm/ieee754-sf.S" line="691" maxcount="6">
  <loop source="gcc-4.4.2/.*/arm/ieee754-sf.S" line="744" maxcount="23">

Bounds seeking

- Demand-driven: call OTAWA’s *mkff* to identify necessary bounds
- Customizable: use/use not pragmas or ORANGE info
  allows to check whether pagai is able to find bounds on its own
Bounds seeking: detailed work-flow and options

Running example:

- no arm-lib bounds (no floating points)
- user-pragma & ORANGE agree on the unique loop bound (10)
Detailed work-flow and options

Backend: owcet + pagai + compare

- ref. BIN
- fixed.ffx
  - otawa's owcet
    - owcet.lp
      - lp_solve
      - wcet 1
  
- ref. C+counters
  - pagai
    - ref. C+counters
      - + invariants
        - pagai2lp
          (retrieve & translate invariants)
        - pagai.lp
          - lp_solve
          - wcet 2
        
- counter/BB
  - option = strategy
    - simple, path foc., etc.

15/26
Running example

- **raw pagai invariants:**
  - \(-10 + \text{cptr\_main\_2} = 0\)
  - \(-10 + \text{cptr\_main\_4} = 0\)
  - \(5 - \text{cptr\_main\_3} \geq 0\)

- **translated into BB ilp constraints:**
  
  \[
  \begin{align*}
  x_4_{\text{main}} &= 10; \quad \text{// already given/found by user/\text{O\text{RANGE}}} \\
  x_6_{\text{main}} &= 10; \quad \text{// structural consequence} \\
  x_5_{\text{main}} &\leq 5; \quad \text{// new information}
  \end{align*}
  \]

- **Final result:**
  
  Estimation WITHOUT PAGAI: 1640
  Estimation WITH PAGAI: 985
Playing with options

Inlining

- deeply changes the program ...
- ... but mandatory for exploiting *pagai* full power:
  - no inter-procedural support for now...
  - ... then *pagai* is unable to relate *caller counters with callee counters*.
  - Inlining is just a “cheat” to see what an interproc-*pagai* would do...

Bounds seeking

- with/without ORANGE/pragmas
- allows to check the ability of *pagai* to find bounds
Optimization level

• one can try standard optimizations O1, O2, but:
  ➔ traceability may be lost (too bad, but safe)
  ➔ traceability may be false (unsafe !)

• However, optimized code can be 3,5,10 times ...
  is it reasonable to forbid optimization ?

• The reasonable solution: traceability-aware compilation
  but requires a lot of work!

• Empirical solution:
  ➔ data-flow optimizations are those that strongly speed-up code ...
  ➔ ... and they don’t strongly damage traceability
  ➔ control-flow optimizations have less influence ...
  ➔ ... so why not forbid them.

  ➔ Is there some ideal, customized -O1 level, that speed up the program without
  modifying the control structure ?
Customized O1 level

- Empirically:
  -O1 -fno-auto-inc-dec -fno-cprop-registers -fno-dce -fno-defer-pop
  -fno-dse -fno-guess-branch-probability -fno-if-conversion2
  -fno-if-conversion -fno-inline-small-functions -fno-ipa-pure-const
  -fno-ipa-reference -fno-merge-constants -fno-split-wide-types
  -fno-tree-sra -fno-tree-ter -fno-unit-at-a-time -fno-crossjumping
  -fno-if-conversion -fno-if-conversion2 -fno-jump-tables -fno-loop-block
  -fno-loop-interchange -fno-loop-strip-mine -fno-move-loop-invariants
  -fno-reorder-blocks -fno-reorder-blocks-and-partition
  -fno-reschedule-modulo-scheduled-loops -fno-unroll-loops
  -fno-unroll-all-loops -fno-unsafe-loop-optimizations -fno-unswitch-loops

- WARNING: not fully tested, just promising!

- Not sure at all it’s minimal: deserve more work

- And moreover, valid only for this particular version of arm-elf-gcc
Running example

<table>
<thead>
<tr>
<th>optim</th>
<th>cfg modif</th>
<th>owcet</th>
<th>+pagai</th>
<th>why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0</td>
<td>no</td>
<td>1640</td>
<td>985</td>
<td>pagai cuts 5 heavy iterations, both find 10 total iterations</td>
</tr>
<tr>
<td>-O1</td>
<td>yes</td>
<td>780</td>
<td>711</td>
<td>pagai cuts nothing, owcet overestimate iterations (11)</td>
</tr>
<tr>
<td>-O2</td>
<td>yes</td>
<td>unb.</td>
<td>694</td>
<td>pagai cuts nothing, owcet miss loop bound</td>
</tr>
<tr>
<td>-C01</td>
<td>no</td>
<td>666</td>
<td>426</td>
<td>pagai cuts 5 heavy iterations, both find 10 total iterations</td>
</tr>
</tbody>
</table>

A (very) preliminary conclusion:

- C-line based ffx mechanism does not support loop transformation:
  - here a ”while do” to ”do while” transformation leads to over-approximation (safe)
  - but what about more complex transformation?
- pagai “seems” safer:
  - does not rely on the loop structure: only on control-points
  - as far as debug info is non ambiguous, the result (should be) safe...
  - ... but traceability may be lost.
- the –C01 is (by far) the best solution:
  - does not impact the ORANGE/owcet interaction,
  - allows pagai to trace interesting information
Some experiments

Benchmarks

- Sequential TacleBench
- Ad-Hoc programs
- Lustre/SCADE programs
- Analysed function: generally main, inlined
- Expected results
  - WCET enhancement w.r.t OTAWA+oRange WCET
  - loop bounds computation
Observed enhancement

- Unused code
  - Statically computable tests
  - Break in an “if”, in a “while”
  - Why ? Cause most of TacleBench are single execution programs!
- Conflicts (i.e. exclusive branches)
  - without loop : incompatible conditions
  - in loops : only n (heavy) iterations over m (n < m)
Loop bounds (32 TacleBench)

- counters alone found bounds : 16
- oRange and counters are complementary : 1 (duff)
- oRange succeeds and not counters : 10 (mainly nested loops)
- oRange doesn’t survive the rewriting : 5

Not surprising: we know that pagai is not the right tool for finding bounds
## TacleBench and Lustre/SCADE programs

<table>
<thead>
<tr>
<th>Bench</th>
<th>program</th>
<th>imp. $^t$</th>
<th>general features</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dead-code</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB-MRTC</td>
<td>adpcm-encoder</td>
<td>2.25%</td>
<td>Break if while</td>
</tr>
<tr>
<td>TB-MRTC</td>
<td>bsort100</td>
<td>1.97%</td>
<td>Break if while</td>
</tr>
<tr>
<td>TB-MRTC</td>
<td>crc</td>
<td>48.70%</td>
<td>Statically comput.</td>
</tr>
<tr>
<td><strong>Conflicts</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB-MRTC</td>
<td>expint</td>
<td>17.84%</td>
<td>in loops</td>
</tr>
<tr>
<td>TB-MRTC</td>
<td>lcdnum</td>
<td>39.10%</td>
<td>in loops</td>
</tr>
<tr>
<td>TB-MRTC</td>
<td>qurt</td>
<td>0.01%</td>
<td>in loops</td>
</tr>
<tr>
<td>TB-Media</td>
<td>h264dec_ldecode_block</td>
<td>68.83%</td>
<td>in loops</td>
</tr>
<tr>
<td>DSP</td>
<td>startup_fixed</td>
<td>0.01%</td>
<td>without loop</td>
</tr>
<tr>
<td>Lustre</td>
<td>access_4cnt</td>
<td>0.59%</td>
<td>without loop</td>
</tr>
<tr>
<td>Lustre</td>
<td>ite</td>
<td>0.56%</td>
<td>without loop</td>
</tr>
<tr>
<td>SCADE</td>
<td>roll_control</td>
<td>0.11%</td>
<td>without loop</td>
</tr>
</tbody>
</table>

Some experiments
# Simple Ad-Hoc programs

<table>
<thead>
<tr>
<th>program</th>
<th>imp. $^t$</th>
<th>general features</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounded anyway</td>
<td></td>
<td></td>
</tr>
<tr>
<td>condcache.c</td>
<td>25.71%</td>
<td></td>
</tr>
<tr>
<td>ifthen.c</td>
<td>8.00%</td>
<td>no loop,</td>
</tr>
<tr>
<td>infeasible.c</td>
<td>5.56%</td>
<td>tests on integer variables and counters</td>
</tr>
<tr>
<td>max.c</td>
<td>24.81%</td>
<td>generally statically computable</td>
</tr>
<tr>
<td>sou.c</td>
<td>3.09%</td>
<td></td>
</tr>
<tr>
<td>bounded only by oRange</td>
<td></td>
<td></td>
</tr>
<tr>
<td>detec.c</td>
<td>0.06%</td>
<td>nested loops</td>
</tr>
<tr>
<td>bounded both by oRange and by Pagai alone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>even.c</td>
<td>23.12%</td>
<td>loop step2, test on counters</td>
</tr>
<tr>
<td>expint.c</td>
<td>17.84%</td>
<td>obfuscated loop bound</td>
</tr>
<tr>
<td>hachis.c</td>
<td>15.98%</td>
<td>for loop, test on index</td>
</tr>
<tr>
<td>loop1.c</td>
<td>20.90%</td>
<td>for loops, unfeasible tests in loop</td>
</tr>
<tr>
<td>propofake.c</td>
<td>99.88%</td>
<td>while loop, stop on counters * 1000</td>
</tr>
<tr>
<td>bubble.c</td>
<td>8.22%</td>
<td>for loop, tests on integer vars in loop</td>
</tr>
</tbody>
</table>
Conclusion & Perspectives

- Semantic properties strongly influence the precision of WCET
- Semantic properties easier to extract from high level code
- Connexion with low-level is possible using debugging information
  - at least with -o0, -o1 (no big change in the control structure)
  - better compiler cooperation would be welcome
- Clever choice of counters to insert
  - the cost of semantic analysis highly depends on the number of counters
  - it’s useless to separate branches with similar durations
- Challenge for loop bounds:
  - current tools (e.g. ORANGE) are mainly pattern-based
  - program analysis is much less dependent on program structure:
    find a way to deal with nested loops?
- Need for interprocedural semantic analysis (presently, often inlined)