Removing Infeasible Paths in WCET Estimation:
The Counter Method

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

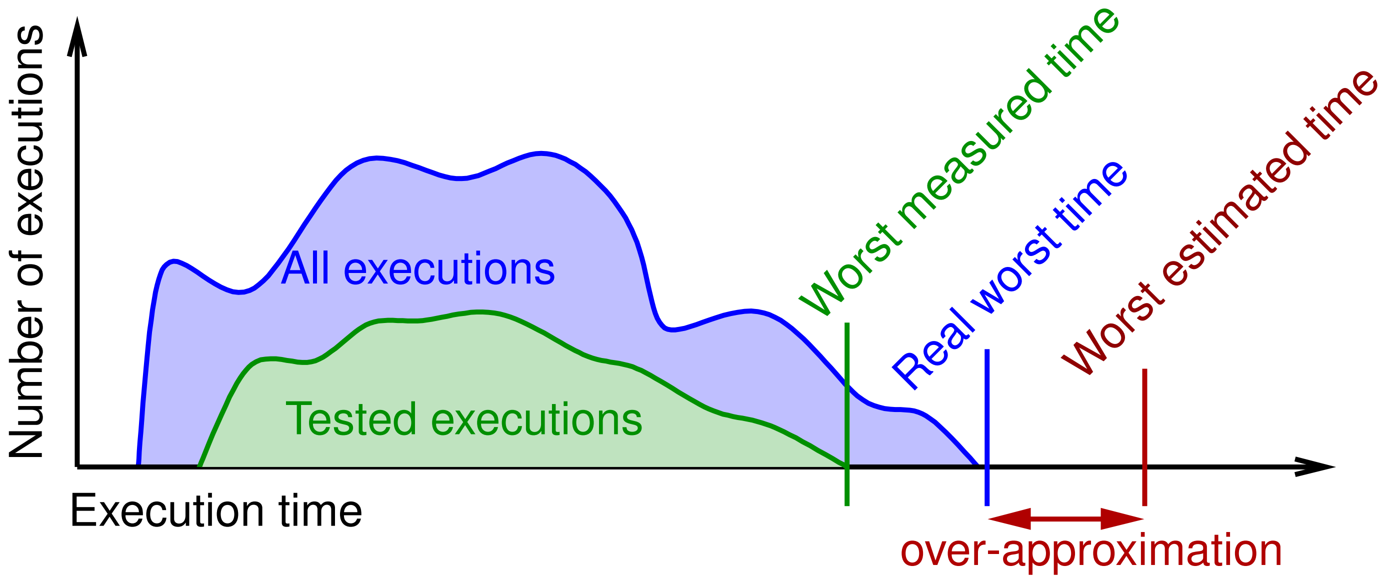
Mihail Asavoae, Rémy Boutonnet, Fabienne Carrier, Nicolas Halbwachs, Erwan Jahier, Claire Maiza, Catherine Parent-Vigouroux, Pascal Raymond

Verimag/Grenoble-Alpes University

SYNCHRON16, dec. 2016, Bamberg

A brief introduction on WCET and IPET

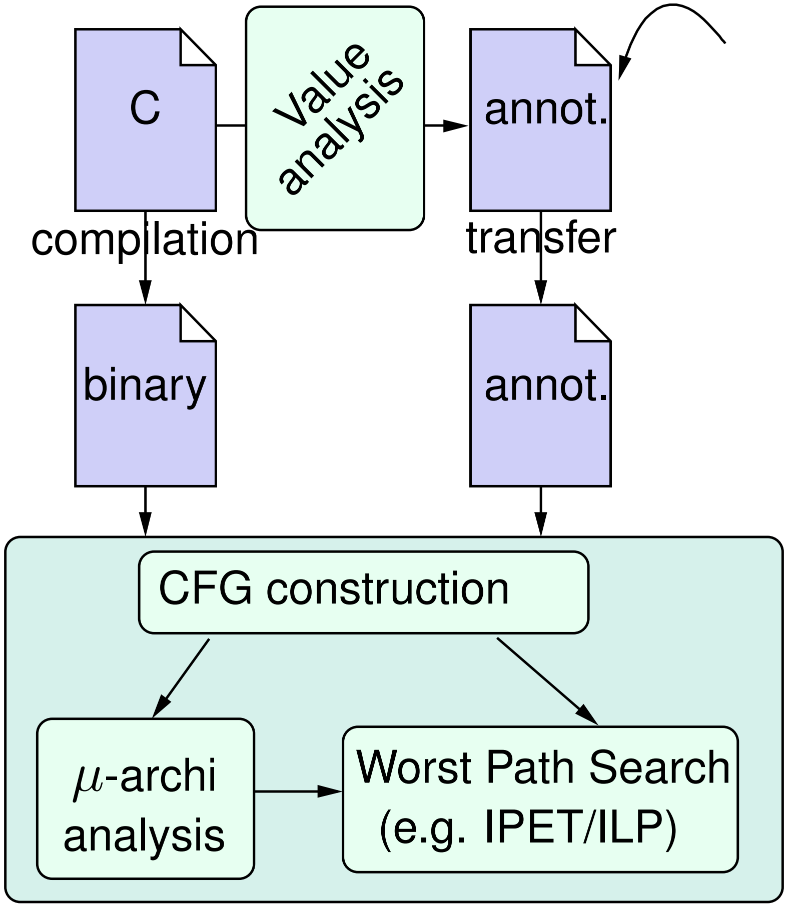
WCET estimation



- 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 over estimated
- Main sources of over-approximation:
  - ↪ Hardware (too complex, abstractions)
  - ↪ Software (infeasible paths)

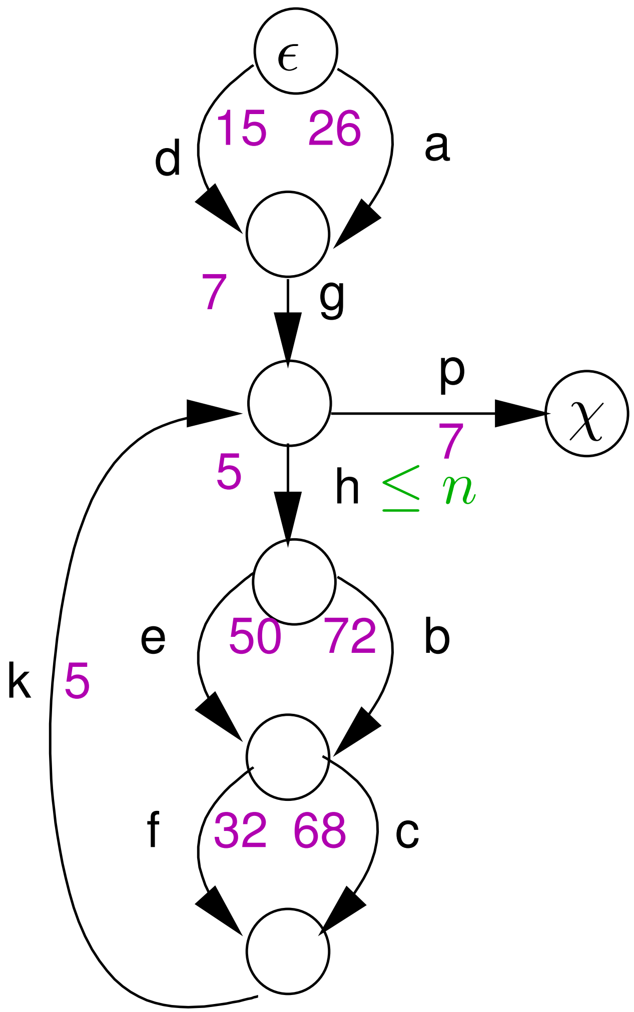
WCET tool organization

- 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

- μ-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: MAX$(\sum_{x \in \mathcal{E}} w_x x)$
  - ↪ 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

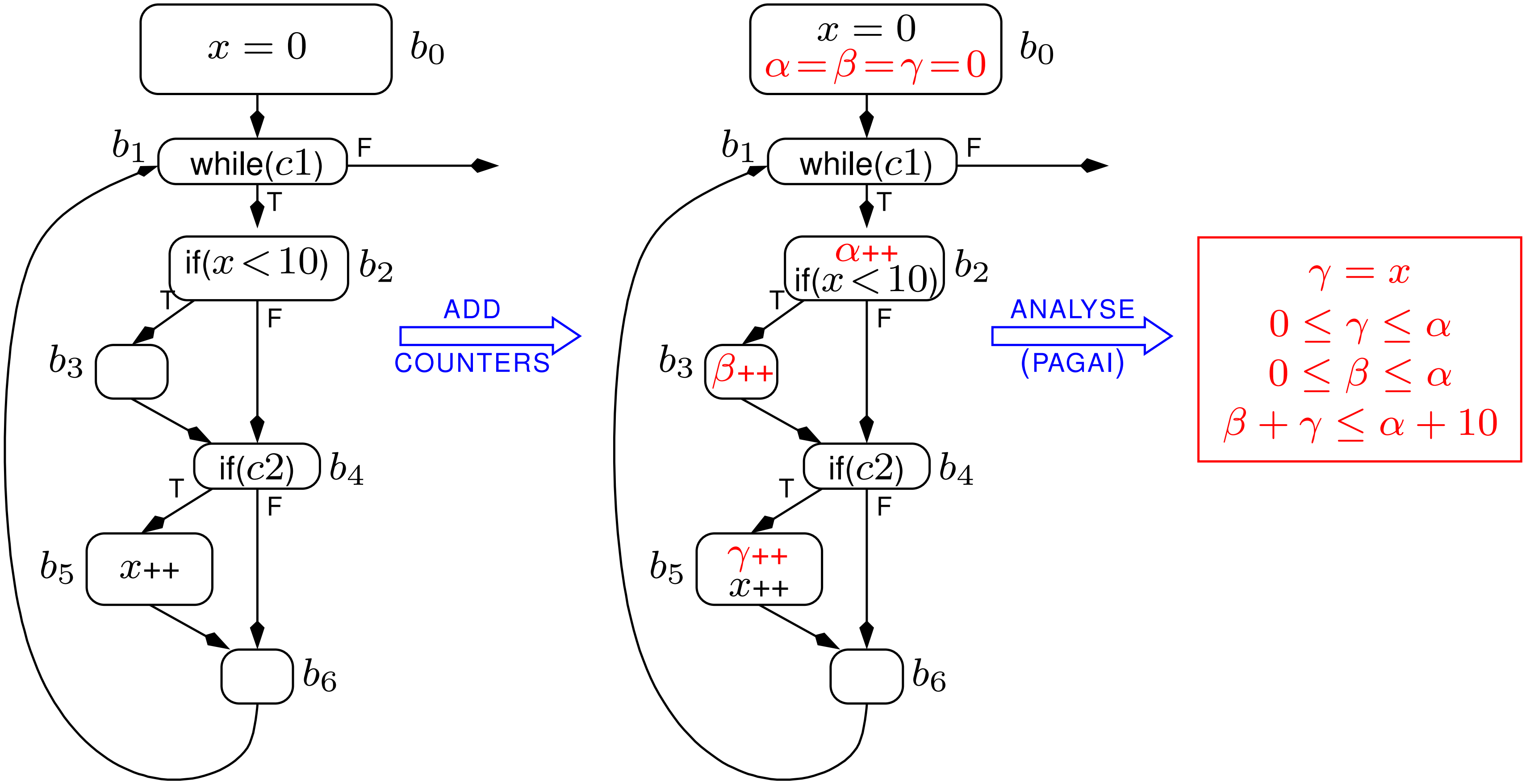
- use *state of the art static analysers* to enhance *state of the art WCET estimation* ...
- ... implies some choices:
  - ↪ program analysis at the C level (that's what program analyzers do...)
  - ↪ comply the IPET/ILP approach (that's what WCET analyzers do...)

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



$$\gamma = x$$
$$0 \leq \gamma \leq \alpha$$
$$0 \leq \beta \leq \alpha$$
$$\beta + \gamma \leq \alpha + 10$$

From principles to practice...

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

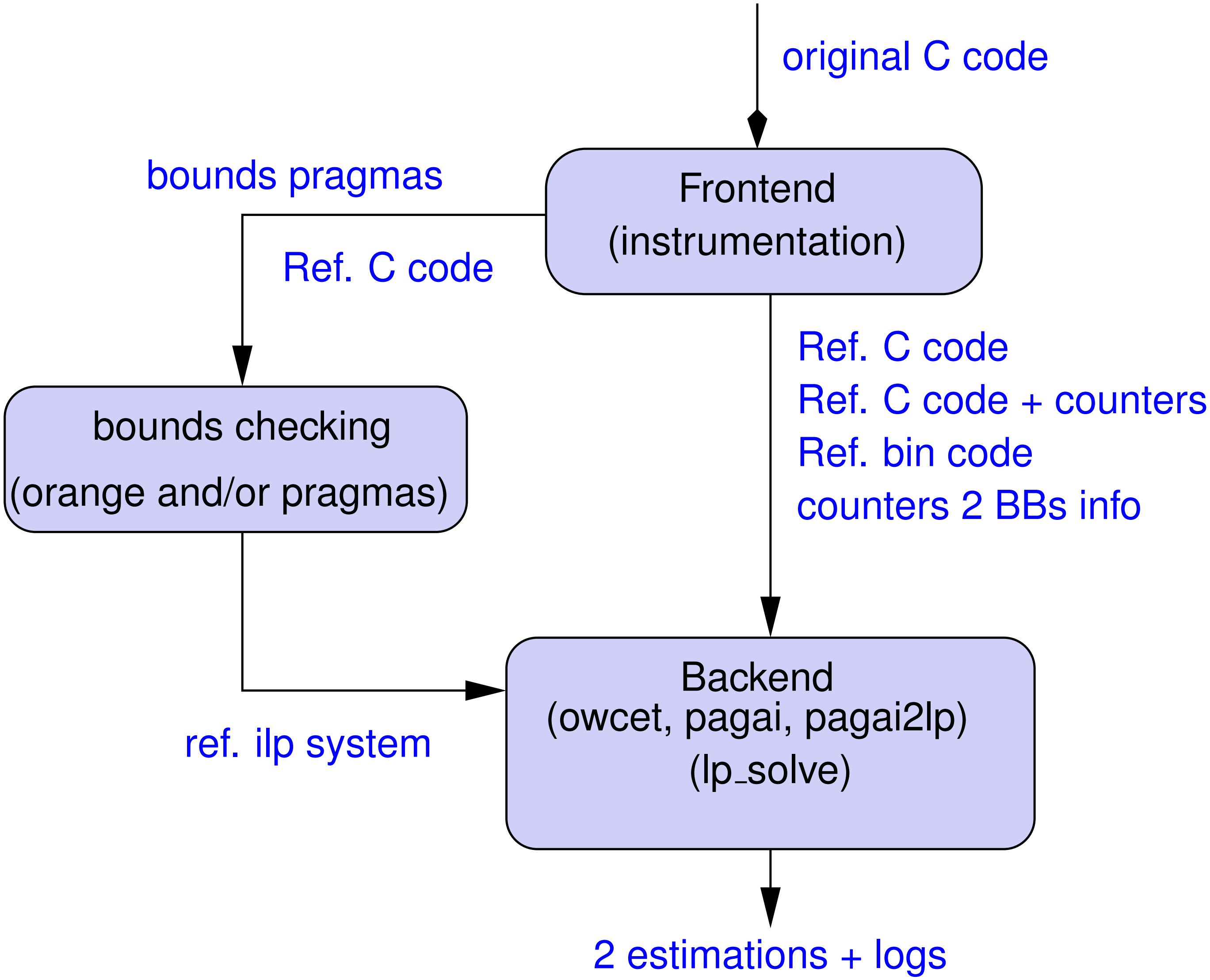
Tools/Technical choices

- OTAWA+`lp_solve` for WCET/IPET and ILP
- `pagai`, (Henry/Monniaux/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



original C code

Frontend
(instrumentation)

bounds pragmas

Ref. C code

bounds checking
(orange and/or pragmas)

Ref. C code
Ref. C code + counters
Ref. bin code
counters 2 BBs info

ref. ilp system

Backend
(owcet, pagai, pagai2lp)
(lp_solve)

2 estimations + logs

Frontend (Instrumentation)

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)

```
#ifdef 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;
}
```

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.

```
int main(void) {
 int i ;
 unsigned char a ;
 unsigned char tmp ;
 int __retres4 ;
 //int cptr_main_1 = 0;
 //int cptr_main_2 = 0;
 //int cptr_main_3 = 0;
 //int cptr_main_4 = 0;
 //int cptr_main_5 = 0;
 //cptr_main_1 ++; #line 144
 i = 0;
 while (i < 10) { //bound=10 #line 146
  //cptr_main_2 ++; #line 147
  a = (unsigned char )IN;
  if (i < 5) {
   //cptr_main_3 ++; #line 150
   a = (unsigned char )((int )a & 15);
   tmp = num_to_lcd(a);
   OUT = (unsigned char volatile )tmp;
  }
  //cptr_main_4 ++; #line 155
  i ++;
 }
 //cptr_main_5 ++; #158
 __retres4 = 0;
#pragma RETURN_BLOCK("main")
 return (__retres4);
}
```

Running example (cntd)

- Reference program is compiled: `lcd_num.elf`...
- ... and counters are associated to (binary) BB, as far as possible:

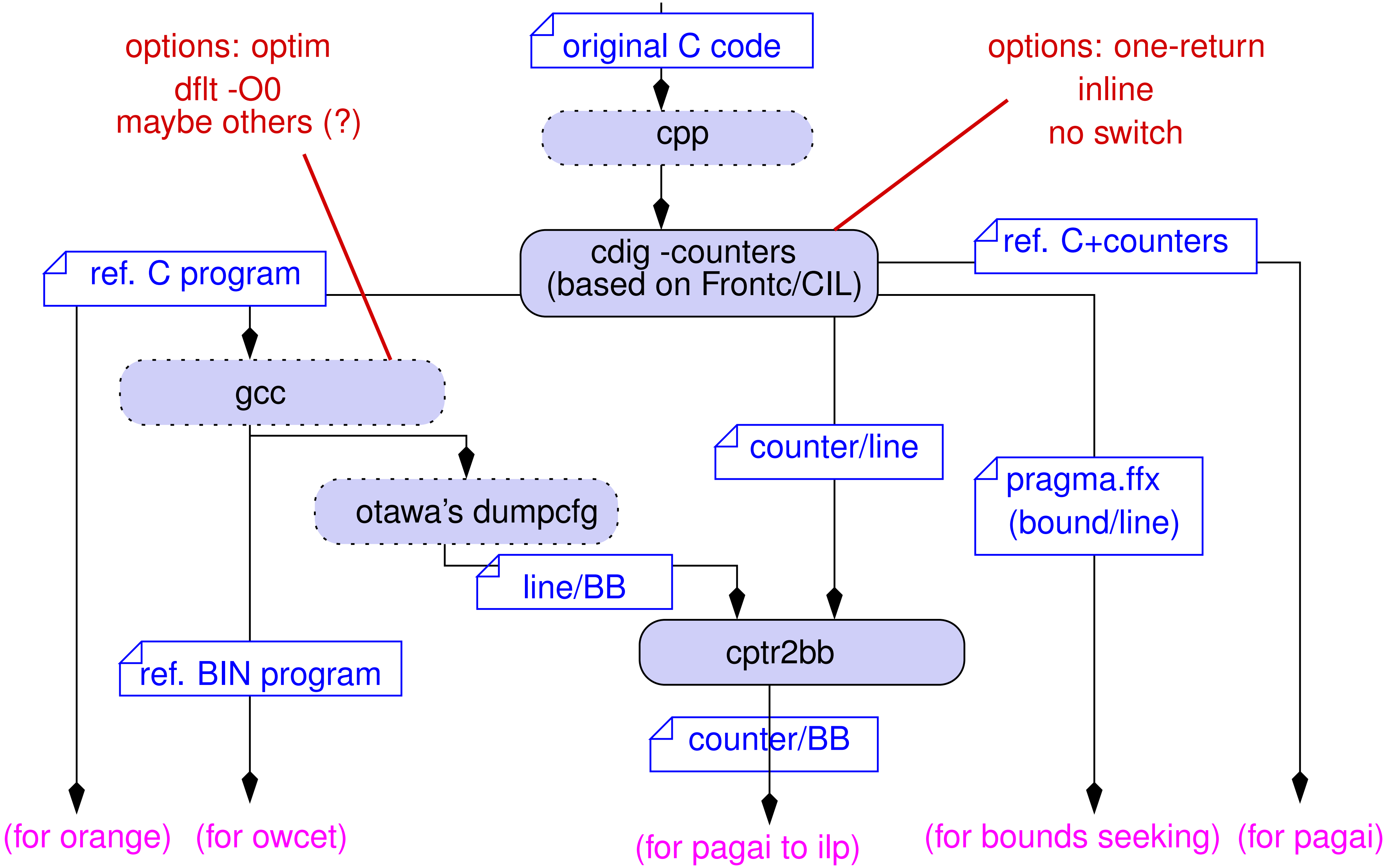
  ↪ we rely on OTAWA's `dumpcfg`, to be sure to agree on BB numbering/source line

  ↪ 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:

line(s)	bloc(s)	reliable	counter
136,144	1	yes	**`cptr_main_1`**
145	1;2	NO	
147,148	4	yes	**`cptr_main_2`**
150,151,152	5	yes	**`cptr_main_3`**
155	6	yes	**`cptr_main_4`**
158,159,160	3	yes	**`cptr_main_5`**

Instrumentation: detailed work-flow and options



options: optim
dflt -O0
maybe others (?)

original C code

cpp

options: one-return
inline
no switch

cdig -counters
(based on Frontc/CIL)

ref. C+counters

ref. C program

gcc

counter/line

pragma.ffx
(bound/line)

otawa's dumpcfg

line/BB

cptr2bb

ref. BIN program

counter/BB

(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.

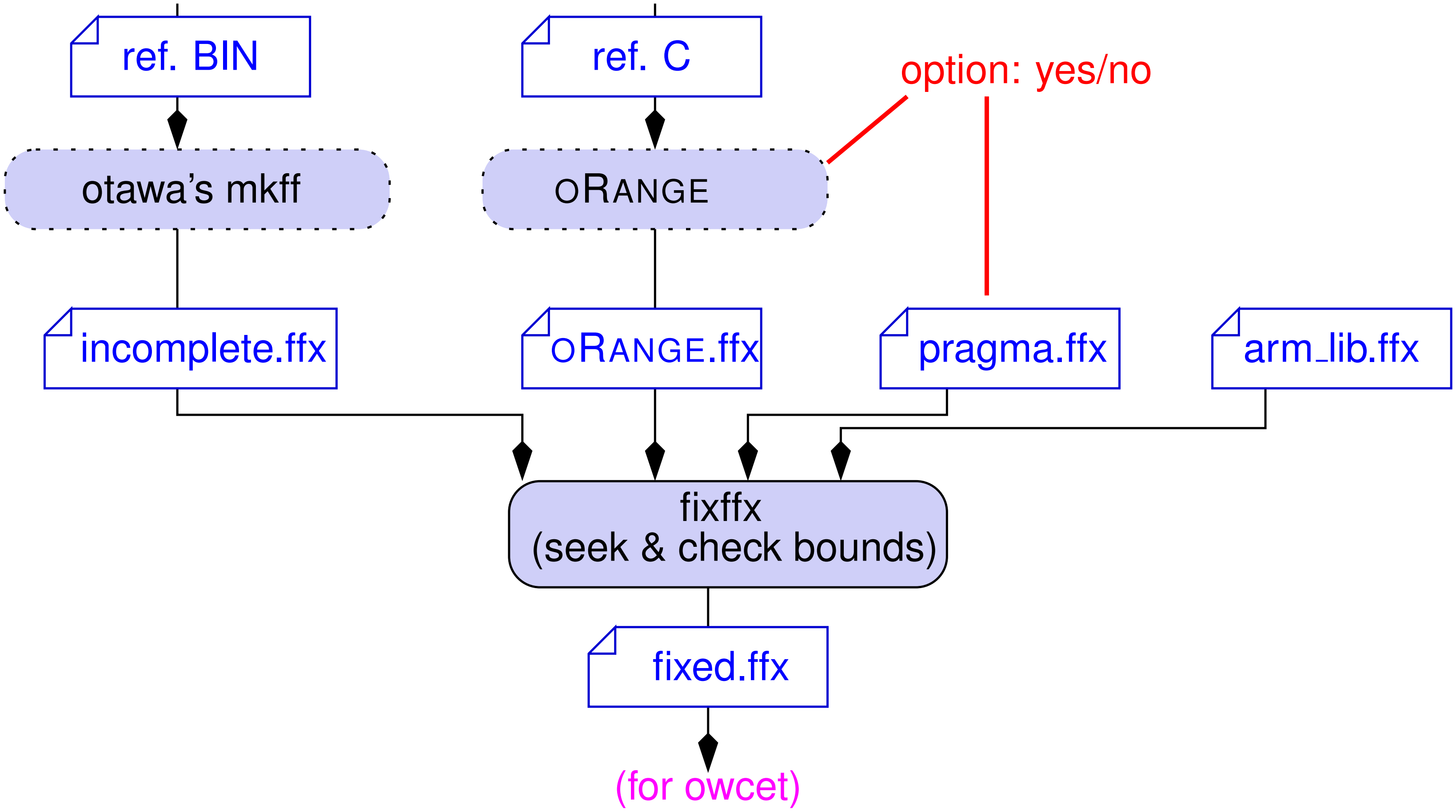
```
<loop source="gcc-4.4.2/.*/arm/ieee754-sf.S" line="691" maxcount="6">
<loop source="gcc-4.4.2/.*/arm/ieee754-sf.S" line="744" maxcount="23">
```

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

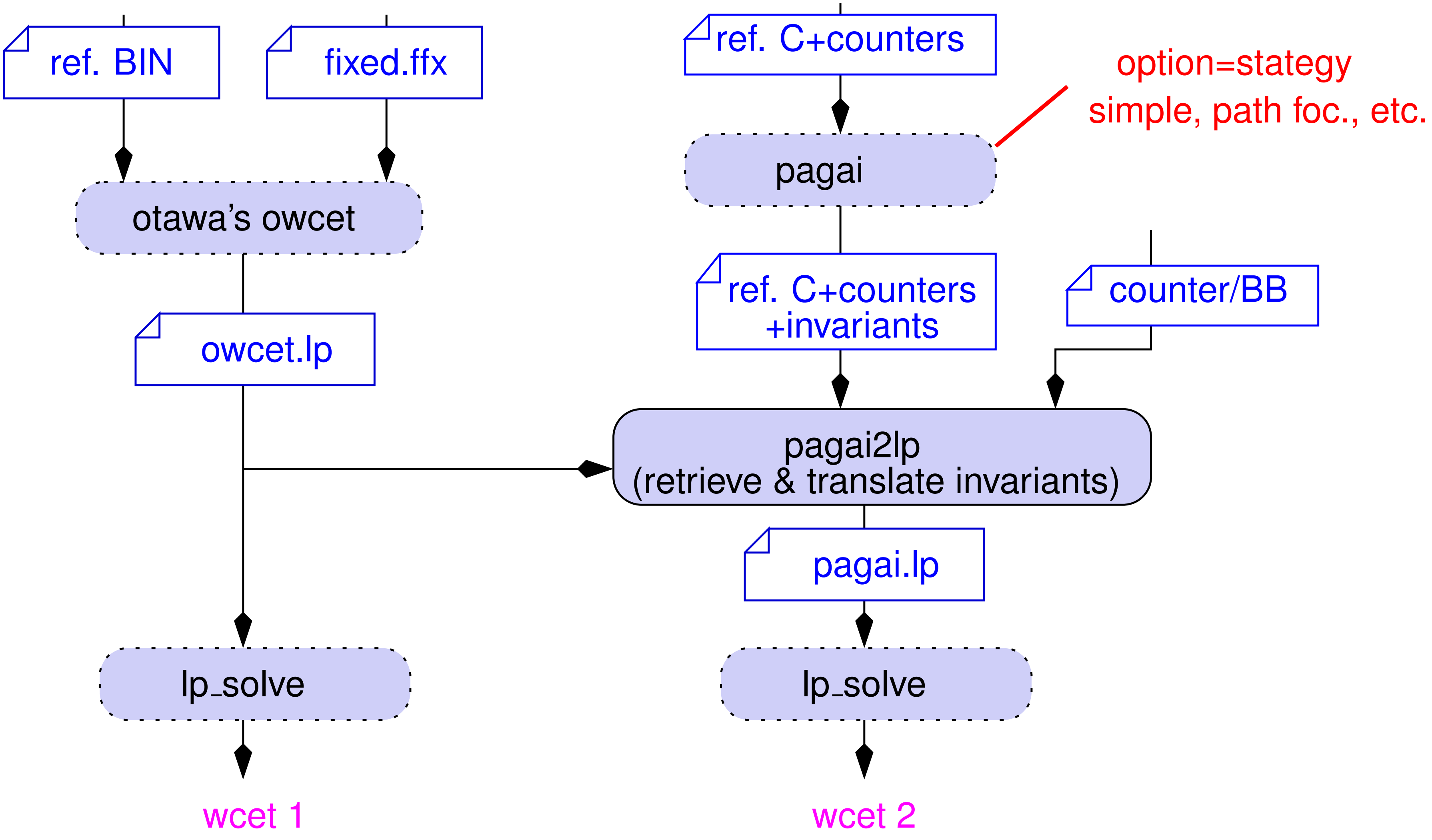


Running example:

- no arm-lib bounds (no floating points)
- user-pragma & oRANGE agree on the unique loop bound (10)

Backend: owcet + `pagai` + compare

Detailed work-flow and options



ref. BIN

fixed.ffx

otawa's owcet

owcet.lp

lp_solve

wcet 1

ref. C+counters

pagai

option=stategy
simple, path foc., etc.

ref. C+counters
+invariants

counter/BB

pagai2lp
(retrieve & translate invariants)

pagai.lp

lp_solve

wcet 2

Running example

- raw `pagai` invariants:

```
-10+cptr_main_2 = 0
-10+cptr_main_4 = 0
5-cptr_main_3 >= 0
```

- translated into BB ilp constraints:

```
x4_main = 10; // already given/found by user/ORANGE
x6_main = 10; // structural consequence
x5_main <= 5; // new information
```

- 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 forbit 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-builtin-call-dce -fno-tree-ccp -fno-tree-ch -fno-tree-copyrename
 -fno-tree-dce -fno-tree-dominator-opts -fno-tree-dse -fno-tree-fre
 -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

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

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 `-CO1` 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

Bench	program	imp.^{t}	general features
Dead-code			
TB-MRTC	adpcm-encoder	2.25%	Break if while
TB-MRTC	bsort100	1.97%	Break if while
TB-MRTC	crc	48.70 %	Statically comput.
Conflicts			
TB-MRTC	expint	17.84%	in loops
TB-MRTC	lcdnum	39.10%	in loops
TB-MRTC	qurt	0.01%	in loops
TB-Media	h264dec_ldecode_block	68.83%	in loops
DSP	startup_fixed	0.01%	without loop
Lustre	access_4cnt	0.59%	without loop
Lustre	ite	0.56%	without loop
SCADE	roll_control	0.11%	without loop

Simple Ad-Hoc programs

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

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)