SPACE SYSTEMS FINLAND

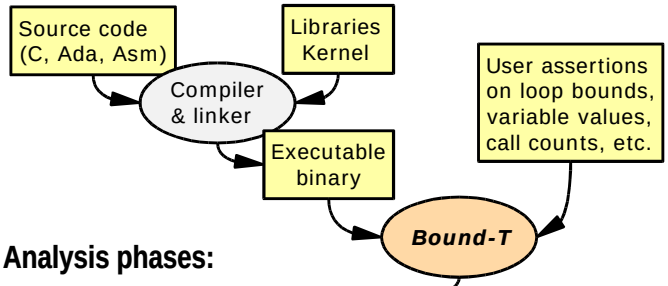
Worst-Case Execution Time Analysis for Digital Signal Processors

Niklas.Holsti @ ssf.fi
Thomas.Långbacka @ ssf.fi
Sami.Saarinen @ ssf.fi

ESTEC contract No. 13362/99/NL/FM

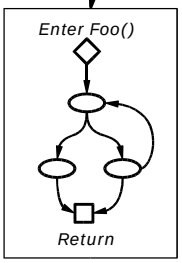
Bound-T tool

Statically analyses an executable binary program, gives bounds on worst-case execution time (WCET)

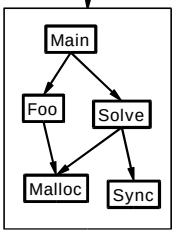


Analysis phases:

- Decode instructions
- Analyse control flow
- Analyse subprogram calls
- Bound loop iterations
- Find worst-case path



Flow graphs



Call graphs

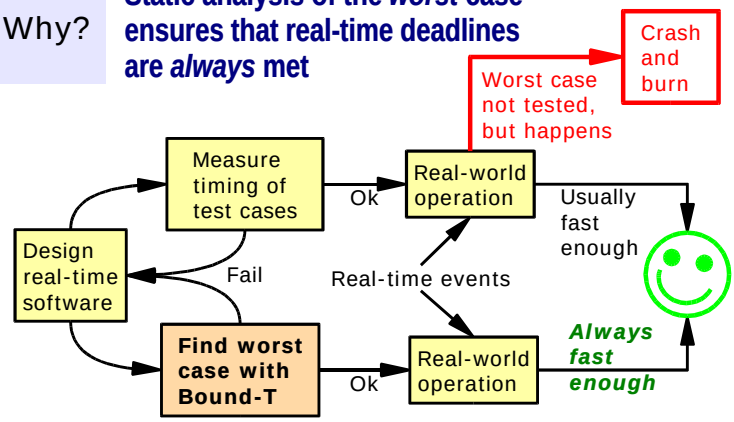
```
Main          345
. Foo         113
. . Malloc     55
. Solve        99
. . Malloc     43
. . Sync       12
```

WCET for calls

Bound-T platforms: Sun Solaris (... Linux, Win NT)
Bound-T targets: ADSP-21020, Intel-8051, ...

Why?

Static analysis of the *worst* case ensures that real-time deadlines are *always* met



How?

Problem is generally unsolvable; made tractable by coding rules and guidance from user-given assertions

Loops should be controlled by ***counters.***
Loop-bounds should be ***static***, or derived from static subprogram arguments.

Methods used in Bound-T include:
Loop bounds: Presburger Arithmetic (Omega System)
Worst case: Integer Linear Programming (lp_solve)

Coded with the GNU Ada Compiler GNAT from ACT.

Loop bounds

Simple, counter loops are bounded automatically; others need assertions

```
#define VEC_LEN 100
float sum_vector (float vec[])
{  int i;
   float sum = 0.0;
   for (i = 0; i < VEC_LEN; i++)
      sum += vec[i];
   return sum;
}
```

Data-flow analysis in Bound-T finds:

Counter is ***i***
Initially ***i = 0***
Finally ***i = 99***
Thus ***100*** repeats.

```
int find (int *vec, int val)
{  int low = 0, high, mid;
   high = VEC_LEN - 1;
   while (low <= high) {
      mid = (low + high) / 2;
      if (vec[mid] == val)
        return mid;
      else if (vec[mid] < val)
        low = mid + 1;
      else
        high = mid - 1;
   }
   return -1;  /* not found */
}
```

No counter; user asserts bounds:

```
file "find.c"
subprogram "find"
loop repeats  <= 7 times;
end_loop
```

DSP particulars

Complex DSP instructions are a challenge for static analysis

Architectural pipelines
Zero-overhead loops
Delayed branches

```
 1 do 9 until ac;
 2   if eq jump 33 (la);
 3   lcntr=50, do 7 until lce;
 4     if eq jump 7;
 5     r5 = r0 + 1;
 6     r6 = r0 - 1;
 7     r7 = r5 * r6;
 8   r8 = r7 + r5;
 9   r9 = r5 + r8;
10 r10 = r0;
```

Example:
One **instruction expands to** ***four*** **flow-graph steps**