EmBounded

WCET Analysis

Reinhold Heckmann

Christian Ferdinand

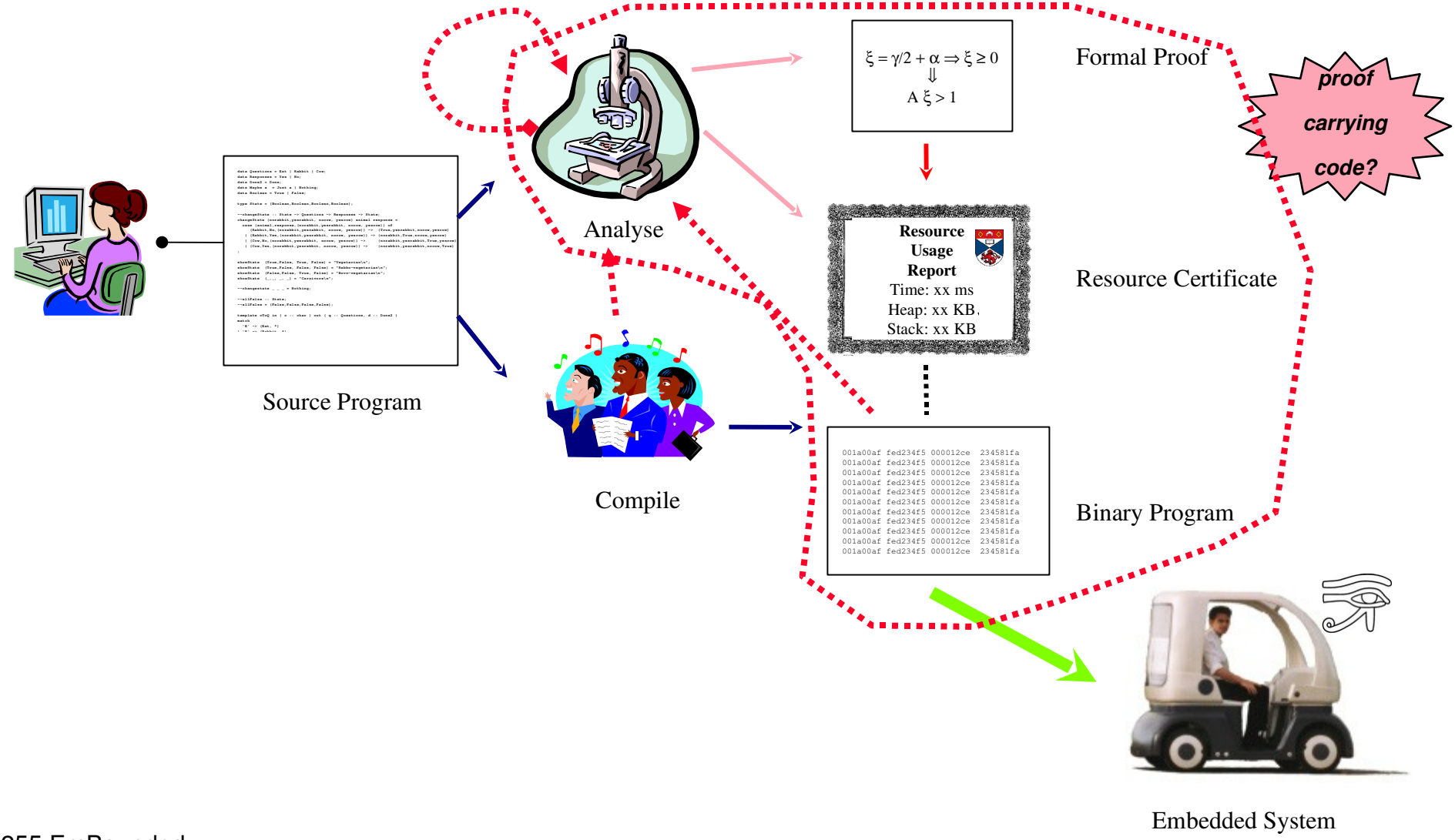
AbsInt Angewandte Informatik GmbH

EmBounded

Overview

- Project Context
- M32
- Analysis of HAM Instructions
- Preliminary Results
- Summary

The EmBounded Vision

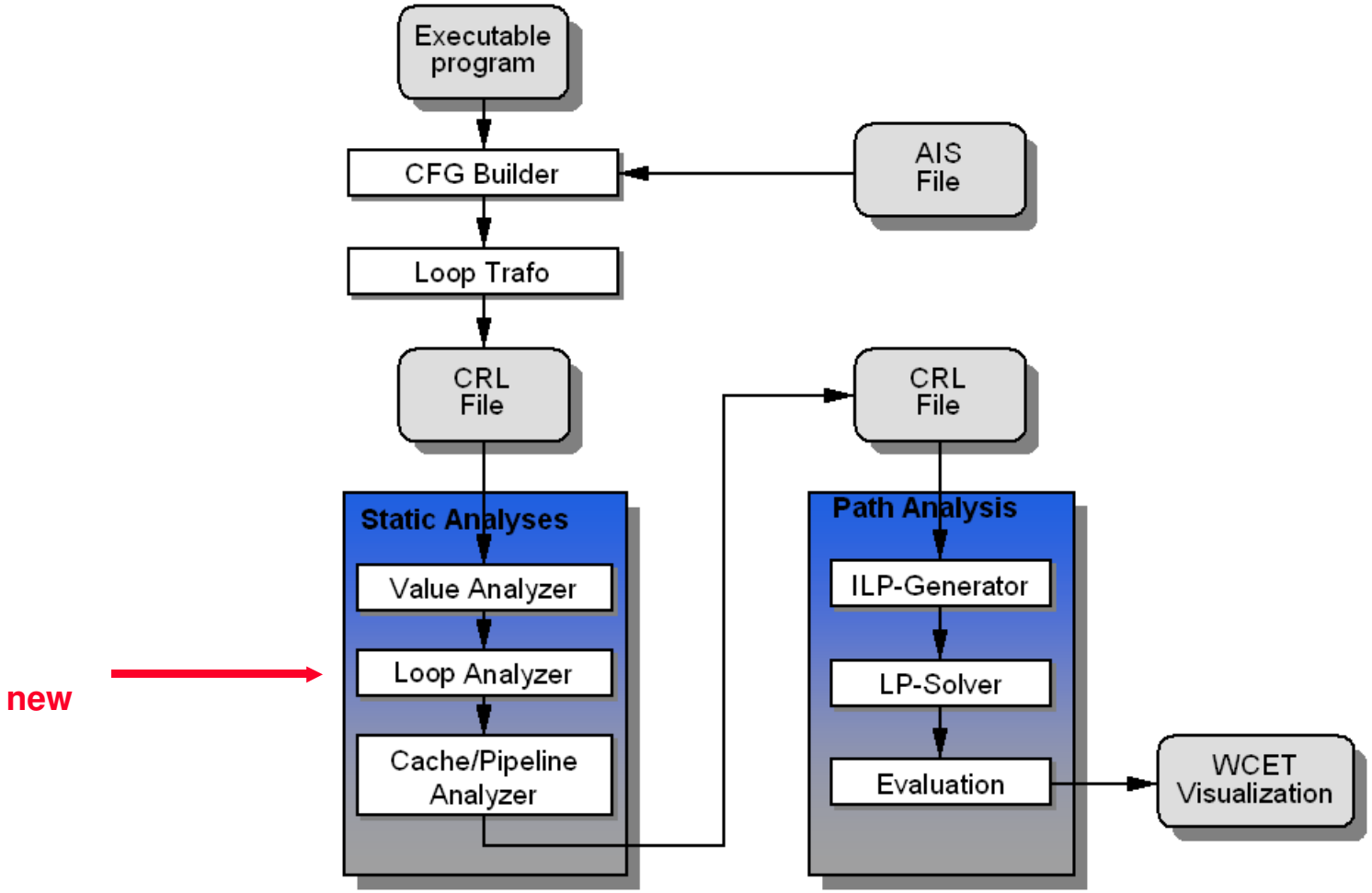


EmBounded

Renesas M16C/M32C

- M32 was selected early as Embounded target
  - Not too complex
  - Availability
  - Good previous experience with Renesas M16
- M16C Family:
  - 3 Pipeline Stages (Fetch/Decode/Execute)
  - 8 Entry Instruction Queue Buffer
  - Various Addressing Modes:
    - **Memory to Register (also Stack Pointer relative)**
    - **Register to Register**
    - **Register to Memory (also Frame Buffer relative)**
    - **Memory to Memory**
- M32C/80 Family:
  - 16/32 bit Instruction Set (CISC & RISC „Harmony“)
  - 108 Basic Instructions
  - Zero, Short, Quick and Generic Instruction Formats
  - High Level Instructions, e.g. RMPA (Repeat MultiPle & Addition)
  - Up to 512 kB Rom and 31 Kb Ram
  - Tailored to automotive applications

aiT WCET Analyzer Structure



Input/Output

Application Code

```
void task
(void)
{
variable++;
function();
    next++;
    if (next
this;
terminate()
}
```

Project file (*.apf)

```
AIS File=hami0111.ais
Additional Executables=
Analysis Control=0
```

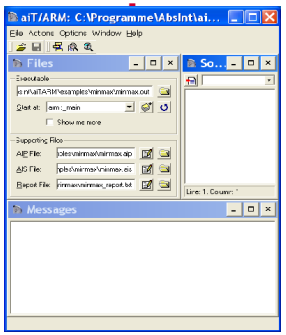
Specifications (*.ais)

```
clock 10200 kHz ;
loop "_codebook" + 1 loop exactly 16 end ;
recursion "_fac" max 6;
SNIPPET "printf" IS NOT ANALYZED AND TAKES MAX
333 CYCLES;
flow "U_MOD" + 0xAC bytes / "U_MOD" + 0xC4 bytes
is max 4;
area from 0x20 to 0x497 is read-only;
```

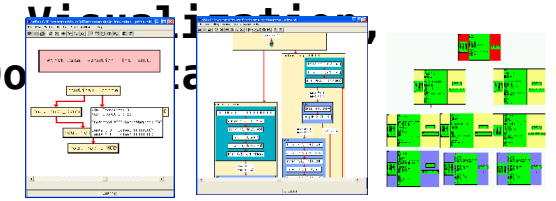
Compiler Linker

Executable (*.elf / *.out)

Entry Point



- Worst Case Execution Time
- Visualization, Do



EmBounded

Extension of aiT for M32

- Value analysis
- Loop Bound Analysis (new)
- Bounds for register-dependent loop instructions
- Extension of the AIS specification language

Value Analysis

- Value analysis tries to find lower and upper bounds for the values in the registers and memory cells
- Value analysis for M32 was implemented during the first year of the EmBounded project

Usage of Value Analysis

The results of value analysis are used to get

- Addresses of memory accesses
  - Important in case of different memory areas with different access times
- Infeasibility information
  - If conditions evaluate always to true or always to false, the unreachable successor need not be analyzed further
- Upper bounds for the iteration numbers of simple loops
  - Loop bounds are derived from initial value of loop counter, from its increment, and from the loop test
  - Newly implemented in Summer 2006

EmBounded

Looping Instructions

- Certain M32 instructions perform a data-dependent loop when executed.
- rmpa, sinb, smovb, smovf, sout, and sstr:
  the number of loop iterations is given by the contents of register R3.
  - Value analysis can find loop bounds for these instructions.
- smovu and scmpu iterate through the memory till they find a zero-byte.
  - aiT does not determine the iteration counts for any of these two instructions. Hence, all these instructions require a user annotation.

AIS Annotations

Generally important annotations:

- INSTRUCTION … CALLS …;
  INSTRUCTION … BRANCHES TO …;
- LOOP *ProgramPoint Qualifier Bounds* ;
  - loop “R” + 10 bytes end max 10;
  - loop “R” + 2 loops begin exactly 20;
- RECURSION "*name*" *Bounds* ;
  - recursion “cmpPrim” max 8;

EmBounded

Annotations specific to M32

- Memory map is fixed apart from
  - End of Internal RAM Block starting at 0x000400:
    global "XXXXXX" = *address* ;
  - Beginning of Internal ROM Block ending at 0xFFFFFF:
    global "YYYYYY" = *address* ;
- Declaration of external bus areas:
  AREA FROM … TO … ACCESS TIME = …;
  - area from 0x80a000 to 0xcfffff access time = 3;

Annotations specific to M32 (cont.)

- For instructions performing data-dependent loops:
  INSTRUCTION … FEATURES
  transfer_count = … ;
  - instruction “R” + 10 bytes features transfer_count = 5;
- Always necessary for smovu and scmpu that loop till a zero-byte is found.
- Necessary for rmpa, sinb, smovb, smovf, sout, and sstr, which depend on R3, if value of R3 is not found by value analysis.

Analysis of HAM Instructions

EmBounded

Example: Simple Simon translated from Hume to HAM to C and then further to M32

- Contains 220 HAM instructions
- Goal: To analyze as many of these instructions as possible

EmBounded

Getting Start Addresses

- Start aiT with given M32 code
- Use entry point `engineRun`
- Compute control-flow graph and disassembly with source code comments
- Special comments indicate the beginning of HAM instructions
- The first code address behind such a comment is the start address of the HAM instruction
  - `/* -------------- 385: MkInt 64 */`
  - `stack[inc_sp()].hp = mkInt(64);`
  - `0xffa57f: mov.l:g 0x648, r3r1`
  - `0xffa583: mov.l:g r3r1, r2r0`

EmBounded

Getting Start Addresses

- Beware: Some HAM instructions have no M32 code

```
-     /* -------------- 88: Goto "n_s_2_0" */
-     goto n_s_2_0;
- 0xffae7e: jmp.w #0xffa69c <0xffa69c>
- n_s_2_0:
-     /* -------------- 96: Goto "s_exit" */
-     goto s_exit;

- s_3:
-     /* -------------- 103: Extension.AvailSet 0 2 */
-     if(!wire[3].availableFlag || !wire[1].availableFlag) {
- 0xffae81: cmp.b:q #+0, 0x6b7
```

EmBounded

Setting up Analyses

- For each HAM instruction generate a project file specifying the start address
  - `Source File=hami.X30`
  - `Stack Address=0x809820`
  - `Start At=0xffaf50`
  - `Strip Compilation Dir=0`
- … and an AIS file specifying the end addresses
- end addresses are the start addresses of all other HAM instructions
  - `end 0xffaee3;`
  - `end 0xffaf33;`
  - `# end 0xffaf50;`
  - `end 0xffaf6c;`
  - `end 0xffaf72;`

Results

- 202 instructions can be analyzed successfully
- 18 instructions cannot due to unbounded loops

Observations:

- (Small) jitter between instructions of the same kind
  - **12: MatchAny 0xffa7f2 10**
  - **37: MatchAny 0xffa8db 11**
  - **64: MatchAny 0xffab07 11**
  - **104: MatchAny 0xffaea3 11**
  - **129: MatchAny 0xffb023 11**
  - **169: MatchAny 0xffb3bf 11**
  - **194: MatchAny 0xffb553 11**
  - **220: MatchAny 0xffb708 10**
  - **254: MatchAny 0xffba05 11**

EmBounded

Dependence on Parameters

- 313: Extension.AvailSet 0 0xffbe85 7
- 327: Extension.AvailSet 1 0xffbe8b 9
- 341: Extension.AvailSet 2 0xffbe94 14
- 11: Extension.AvailSet 0 2 0xffa7d0 16
- 36: Extension.AvailSet 0 2 0xffa8b9 15
- 63: Extension.AvailSet 0 2 0xffaae5 15
- 103: Extension.AvailSet 0 2 0xffae81 15
- 128: Extension.AvailSet 0 2 0xffb001 15
- 168: Extension.AvailSet 0 2 0xffb39d 15
- 193: Extension.AvailSet 0 2 0xffb531 15
- 219: Extension.AvailSet 0 2 0xffb6e6 16
- 248: Extension.AvailSet 1 2 0xffb93b 15
- 288: Extension.AvailSet 1 2 0xffbd19 248

- Last one escapes to `__humeSuperstep`
  I.e. code not belonging to any HAM instruction

Problems with Loops

– 51: CallPrim "*"	0xffa9fe	261	no loops
– 79: CallPrim "+"	0xffac49261	two analyzable loops	
– 85: CallPrim "=="	0xffadd8	?	2 loops + scmpu.b
– 84: CallPrim "mod"	0xffad77	?	4 unbounded loops
– 17: Unpack	0xffa832?	1 unbounded loop	
– 29: CheckOutputs	0xffa69c?	6 unbounded loops	
– 387: Write 1	0xffa5d7	?	6 unbounded loops

Summary

- aiT for M32 implemented/adapted
- First experiments give promising results
- Future work: Communicating loop bounds
- How to compute global WCET