Embedded Program Annotations for WCET Analysis

Bernhard Schommer[1] Christoph Cullmann[2] Gernot Gebhard[2] Xavier Leroy[3] Michael Schmidt[2] Simon Wegener[2]

[1]Saarland Informatics Campus
Saarland University

[2]AbsInt Angewandte Informatik GmbH

[3]INRIA Paris

18th International Workshop on Worst-Case Execution Time Analysis

The work presented here has been conducted within the European ITEA3 project ASSUME (project number 14014), supported by the German Federal Ministry for Education and Research with the funding ID 01IS15031B and the French Ministry for the Economy and Finance. The responsibility for the content remains with the authors.

Safety-Critical Hard Real-Time Software

- Controllers in planes, cars, plants, ... are expected to finish their tasks within reliable time bounds.
- Timing analysis must be performed.
- Safety standards like ISO-26262 require an upper estimation of the execution time for higher criticality levels.

Two Levels of Timing Analysis

- Code level: Single process/task/ISR.
  Focus on control-flow, processor architecture
  incl. pipelines and caches, WCET.
- System level: Multiple processes/tasks/ISRs.
  Focus on integration and scheduling, end-to-end timing,
  worst-case response time (WCRT).

aiT

State-of-the-art industrial-strength static WCET analysis tool.

- Based on Abstract Interpretation
- Works on the binary level.
- Computes safe upper bounds of the worst-case execution time.

AIS Annotation Language

Analysis in aiT can be guided by AIS Annotations:

- Characterisitics of the target hardware.
- Additional semantic information about inputs, memory state, etc.
- Specification of loop bounds, etc.

However, all these annotations relate to the machine code level!

AIS Annotations and DWARF debug information

aiT can extract AIS annotations from the from source code via special markers in C comments, but:

- Only feasible when the source code and debugging information is available.
- Relies on line and column information, which is problematic for Macro usage and ignores preprocessor directives.
- Allows mismatch if source and binary version don't match.

Example

```
double func (double x){
    double data [10] = { ... };
    // x is known to be always >= 0.0
       ↪ and < 10.0
    int i = x ;
    // Refine the value in the location
       ↪ holding variable i
    return data [i]; // Access to
       ↪ unknown address
}
```

CompCert

A proven correct optimizing compiler, usable for critical embedded software.

- Source language: a very large subset of C99.
- Target language: PowerPC/ARM/x86/RISC-V assembly.
- Generates reasonably compact and fast code.

The correctness proof (semantic preservation) for the compiler is entirely machine-checked, using the Coq proof assistant.

Builtin AIS Annotations

Automatic mechanism to transfer annotations from source code to machine code level:

- Printf like syntax `__builtin_ais_annot("...",...)`
- String argument contains the AIS annotation.
- Can contain format specifiers like `%here` for address of current program point and `%e<n>` for AIS expression relating to optional arguments.

Builtin AIS Annotations

Automatic mechanism to transfer annotations from source code to machine code level:

- Annotations are treated like functions calls to builtin functions.
- Annotations are placed in special ELF-section and extracted by aiT.
- Uses standard assembler and linker features only.
- CompCert is completly agnostic towards the annotation content.

Example: Source Code

```
double func (double x){
    double data [10] = { ... };
    // x is known to be always >= 0.0
       ↪ and < 10.0
    int i = x ;
    // Refine the value in the location
       ↪ holding variable i
    __builtin_ais_annot("try instruction
       ↪  %here { enter with : %e1 =
       ↪ 0..9; }; ",i);
    return data [i];
}
```

Example: Assembly

```
; __builtin_ais_annot("try instruction %
    ↪ here { enter with: %e1 = 0..9; };"
    ↪ , i);
.L118:
...
.section  "__compcert_ais_annotations",,
    ↪ n
.ascii "# file:test.c line:25 function:
    ↪ func\ntry instruction "
.byte 7,4
.4byte .L118
.ascii " \{ enter with: reg("r5") =
    ↪ 0..9; \};"
```

Example: Extracted Annotation

```
# file:test.c line:25 function:func
try instruction 0x10013c { enter with:
    ↪ reg("r5") = 0..9; };
```

Preservation Guarantees for Annotations

CompCert treats `__builtin_ais_annot()` as observable event that results in strong guarantees:

- Annotations are only removed if unreachable.
- Order of annotations is preserved.
- Values of arguments are preserved.

Validation of Linking and Assembling

What happens if the Linker or Assembler is buggy?

- Valex tool for linking and assembling validation.
- Checks that annotations are contained in the binary.

Interactions with Compiler Optimizations

Semantic preservation gives strong guarantees but does not prevent any of the following:

- Moving of annotations around pure computations. (This is, however, not done at the moment).
- Removing of symbols referenced in annotation text, but not in arguments.
- Call annotations can become invalid due to inlining/tail-call optimizations.
- Annotations containing specifications for other program points may become unreachable.

Example: Reference of Symbol

```
__builtin_ais_annot ("...static_var..."
    ↪ ); // risky
__builtin_ais_annot("...%e1..." , &
    ↪ static_var); // safe
```

Example: Unreachable Annotation

```
int x = 5;
if (x == 7) {
    __builtin_ais_annot("# some global
        ↪ AIS annotation"); // removed
}
```

Loop Optimizations

Loop optimizations may cause problems with loop annotations:

- Unrolling may change loop iteration counts.
- Loop nest optimization may change iteration order.

$\Rightarrow$ Complex problem.

- CompCert would need to incorporate the AIS semantics in order to be allowed to change the annotations (contrary to the current design).
- However, at the moment CompCert performs no loop optimizations.

Future Work

- Transfer internal compiler knowledge in annotations.
- Mechanism for annotating loops in the presence of loop optimizations.
- Mechanism for top-level annotations.

Summary

We have presented `__builtin_ais_annot()`, a mechanism

- to transfer AIS annotations from the source to machine code level,
- robust in the presence of optimizations,
- and with formally proven correctness guarantees.

Questions?