HAL
open science

Embedded Program Annotations for WCET Analysis

Bernhard Schommer, Christoph Cullmann, Gernot Gebhard, Xavier Leroy, Michael Schmidt, Simon Wegener

To cite this version:

Bernhard Schommer, Christoph Cullmann, Gernot Gebhard, Xavier Leroy, Michael Schmidt, et al.. Embedded Program Annotations for WCET Analysis. WCET 2018: 18th International Workshop on Worst-Case Execution Time Analysis, Jul 2018, Barcelona, Spain. 10.4230/OASIcs.WCET.2018.8 . hal-01848686

HAL Id: hal-01848686

https://inria.hal.science/hal-01848686v1

Submitted on 25 Jul 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Embedded Program Annotations for WCET Analysis

Bernhard Schommer
Saarland Informatics Campus
Saarland University Building E1.3, D-66123 Saarbrücken, Germany
schommer@absint.com

Christoph Cullmann
AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
cullmann@absint.com

Gernot Gebhard
AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
gebhard@absint.com

Xavier Leroy
INRIA Paris
2 rue Simone Iff, 75589 Paris, France
xavier.leroy@inria.fr

Michael Schmidt
AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
schmidt@absint.com

Simon Wegener
AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
wegener@absint.com

Abstract

We present `__builtin_ais_annot()`, a user-friendly, versatile way to transfer annotations (also known as flow facts) written on the source code level to the machine code level. To do so, we couple two tools often used during the development of safety-critical hard real-time systems, the formally verified C compiler CompCert and the static WCET analyzer aiT. CompCert stores the AIS annotations given via `__builtin_ais_annot()` in a special section of the ELF binary, which can later be extracted automatically by aiT.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Hardware → Static timing analysis, Software and its engineering → Embedded software, Software and its engineering → Software verification, Software and its engineering → Automated static analysis, Software and its engineering → Compilers

Keywords and phrases Worst-Case Execution Time (WCET) Analysis, Annotation Support, CompCert, Tool Coupling, aiT

Digital Object Identifier 10.4230/OASIcs.WCET.2018.8

Acknowledgements The work presented in this paper has been conducted within the European ITEA3 project ASSUME (project number 14014), supported by the German Federal Ministry

© Bernhard Schommer, Christoph Cullmann, Gernot Gebhard, Xavier Leroy, Michael Schmidt and Simon Wegener;
licensed under Creative Commons License CC-BY
18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018).
Editor: Florian Brandner; Article No. 8; pp. 8:1–8:12
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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.

1 Introduction

For safety-critical hard real-time systems, it is not only crucial that the software computes the correct result, but also that this happens in a timely manner. Examples for such software are the flight control systems of modern aircraft or the crankshaft-synchronized tasks of automotive motor control systems. One must therefore determine the worst-case execution time of critical parts of the software to get an embedded system certified.

The execution time of a program depends on three things: The input data of the program, the state of the CPU core on which the program is executed, and interference from the outside world (e.g. due to preemption or shared resources). It is thus not sufficient to just measure the execution time of a task once from start to end, as this measurement might underestimate the true WCET. Since the determination of the exact WCET is undecidable in general, WCET analysis tools instead compute estimates or safe upper bounds. There exist basically two mechanisms to do this [15, 14]:

- First, there are static analysis techniques that compute safe upper bounds with the help of a mathematical model of the execution platform. The precision of the results mostly depends on the predictability of the used hard- and software architecture [7, 5], but also on the availability and quality of the documentation concerning the execution time. If a feature is not described at all, or not well enough, the analysis model must incorporate the worst possible outcome to ensure soundness. AbsInt's aiT [2] is such a static WCET analysis tool.
- Second, there are measurement-based/hybrid techniques that obtain execution times from measurements on the real hardware. However, their soundness depends on whether it is possible to measure all possible executions of a program, which is usually infeasible due to the huge state space. AbsInt's TimeWeaver [4] is such a hybrid WCET analysis tool.

There exists a quasi-standard architecture for static WCET analysis tools. First, a binary reader disassembles a fully linked binary input executable into its individual instructions. Architecture specific patterns decide whether an instruction is a call, branch, return or just an ordinary instruction. This knowledge is used to form the basic blocks of the control flow graph (CFG). Then, the control flow between the basic blocks is reconstructed. In most cases, this is done completely automatically. However, if a target of a call or branch cannot be statically resolved, then the user needs to write some annotations to guide the control flow reconstruction. On this control flow graph, several static analyses take place to determine the values of registers and memory cells, addresses of memory accesses, bounds of loops and recursions, as well as infeasible code. Sometimes, loop bounds cannot be computed statically. Then, the user can guide the analysis via annotations. With this information, a microarchitectural analysis is started. There, a mathematical model of the target processor is used to derive timing bounds for each instruction, incorporating the intrinsic behavior of the pipeline and the memory hierarchy, in particular the caches. This abstract model does not cover all the features of the concrete processor, but only those that are relevant for timing analysis. Subsequently, the CFG together with the basic block timing information are used to construct an integer linear program (ILP). Solving this ILP gives then a path with the longest execution time (and consequently, an estimate of the worst-case execution time). For a measurement-based/hybrid WCET analysis tool, the microarchitectural analysis is replaced

by an analysis step which extracts the basic block timing information from measurements taken on the real hardware.

The timing analysis tools of AbsInt can be guided via AIS annotations [1]. These annotations need to be specified on the machine code level, referring to e.g. code addresses or memory cells, since the analysis works on the binary level. Usually, they are provided as extra files. The programmer, however, is normally more apt on the source code level. Thus, every tool support to bridge the gap between the (high-level) source code and the (low-level) machine code is welcome. We want to aid the programmer by offering a way to express annotations on the source level which are then automatically transferred to the binary level.

2 Embedded Program Annotations for aiT

In order to provide an automatic mechanism to transfer annotations from the source code level to the machine code level, we extended CompCert's already existing annotation mechanism via `__builtin_annot()` to (a) generate AIS compatible syntax for program points, registers, and memory cells and (b) store the generated annotations in a special section of the ELF executable. Consequently, we named the mechanism `__builtin_ais_annot()`. This mechanism is available in CompCert public version 3.3 and commercial version 18.04, and is supported in aiT version 18.04.

CompCert collects all annotations contained in a compilation unit and stores them in encoded form in a special section of the object file (`__compcert_ais_annotations`, see the assembler listing in the example below). While creating the final executable, the linker collects all annotation sections from the object files, concatenates them, and stores the result in the executable. Since we only use standard constructs of the assembler and linker, no changes neither to the linker nor to the assembler are necessary. To ensure that the final executable contains the special annotation section, the linker must be instructed to keep the section `__compcert_ais_annotations`, e.g. with a linker command file. Like debug sections, the annotation section is marked non-allocated/non-executable so that it is not loaded in the running program. aiT can automatically extract the annotations from section `__compcert_ais_annotations` and utilize them in analyses. Note that the order in which annotations are exported into the final executable is explicitly undefined. It is therefore not possible to rely on a specific order in which aiT will see the annotations.

For CompCert, annotations via `__builtin_ais_annot()` look like a call to a variadic function similar to "printf": The first argument contains the AIS annotation and is also a format string. It can contain format specifiers like `%here` or `%e`, where the latter is tagged with an index number and refers to a specific argument independent of the order. It is also possible to refer to an argument more than once. `%here` is replaced with the absolute address of the annotation location in the final executable. Expressions, i.e., `%e1`, `%e2`, ... are replaced with an AIS expression for the value of the first, second, ... additional argument.

Furthermore, if the argument of the `__builtin_ais_annot()` is a constant pointer, the generated annotation contains the corresponding symbol name. This reference, then, is resolved by the linker to the address of the symbol, which allows to specify ambiguous symbols, for example static variables or functions.

Semantics

CompCert treats `__builtin_ais_annot()` as a call to an external function. No actual code is generated for the call, but the parameters of the builtin will be evaluated, as it is mandatory in C semantics. The execution of a `__builtin_ais_annot()` statement is

Listing 1 Source code annotations as part of dead code.

```
// assume that count is always zero
for (int i = 0; i < count; i++) {
    __builtin_ais_annot("try loop %here bound: %e1;", count);
    ...
}
```

modeled as producing an observable event that includes the text of the annotation and the values of the arguments. CompCert's formal correctness proof guarantees that, for a C source program that is deterministic and free of undefined behaviors, the compiled code performs the same observable events and in the same order as the source code [10, section 2]. This gives strong guarantees that annotations are preserved throughout CompCert's compilation and optimization passes.

If all additional arguments are non-volatile C variables or compile-time constant expressions, it is guaranteed that no additional code will be generated for `__builtin_ais_annot()`. Moreover, the location displayed as a replacement for the `%e` sequence is guaranteed to be the location where the corresponding variable resides.

Note that using local variables in parameter expressions to `__builtin_ais_annot()` may extend the liveness of those variables and hence prevent some optimizations. Furthermore, since `__builtin_ais_annot()` is considered a call to an external function it also acts as a barrier for many optimizations. In the current implementation, `__builtin_ais_annot()` can only be used at places where C statements are valid, i.e. within a function definition.

CompCert has no knowledge about the AIS annotation language and checks neither the syntax nor the semantics of the annotations. aiT can extract either all annotations embedded in an executable or none. Analyses that cover only a portion of the binary code—e.g., when doing a separate analysis for each task of the executable—may therefore issue warnings for annotations of unreachable program points. The `try` keyword of AIS can be used to suppress such warnings.

Robustness

In the context of optimizations and conditional compilation, the builtin is a *robust* mechanism to attach annotations to specific code locations. It does not rely on the rather ambiguous line information of the DWARF debug information, for example in macro usages, but rather utilizes the label mechanism of the assembler and linker to generate annotations with actual code addresses.

With the builtin-mechanism it is possible for CompCert to e.g. remove unreachable code together with the contained annotations or do code transformations like reordering code blocks without breaking the annotations. Consider the C code snippet in Listing 1. If constant propagation can prove that `count` is always zero, CompCert can remove the whole loop since it will never be executed. In such a situation the annotation will also be removed. In contrast to this, a conventional source code annotation via special formatted C comments (e.g. `/* ai: ...*/`) would remain visible and probably cause problems since aiT collects such annotations by scanning the source code without knowledge of any compiler optimizations. The same is true, when source code uses the C preprocessor for conditional compilation: CompCert can remove unused annotations while conventional source code annotations will remain visible for aiT. Section 4 discusses interactions with compiler optimizations in more details.

Listing 2 Small C code 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
    __builtin_ais_annot("try instruction %here { enter with: %e1 =
        ↪ 0..9; };", i);
    return data[i];
}
```

Validation

In order to ensure that the linking was performed correctly, there exist the linker validation tool Valex. It takes as input a dump of the intermediate representation of the abstract assembly syntax as well as the linked binary and validates that the functions contained in the assembly are preserved, the addresses of the symbols are consistent and the initialization data of global variables is correct. In order to validate that all annotations are correctly translated and contained in the linked binary, we extended Valex to also check whether the AIS annotations are contained in the special AIS section `__compcert_ais_annotations` and that the addresses of the symbols used in the annotations are consistent.

3 Examples/Use Cases

The following, we present the different parts of the mechanism on an example. Moreover, we show how to use the annotation mechanism with aiT in, e.g., a software library that will be integrated in an embedded system.

Refinement/Assertion of Values

The first example is borrowed from [6]. In this example, shown in Listing 2, a double value with a known range is converted to an integer and used as an index, e.g. to access a look-up table. aiT has currently no knowledge of floating point values and assumes the full range of possible values for them. Thus, it needs help to restrict the range of `i` (and derived from it, a memory access) to a small range.

CompCert will emit the PowerPC assembly code shown in Listing 3 when compiling the example code. The assembler code at the labels `.L116`, `.L117` and `.L119` corresponds to the C code shown in the assembler comments below the labels. The assembler code at label `.L120` removes the stack frame, whereas the assembler code at label `.L121` performs the actual return. The format specifier `%here` has been replaced by CompCert with the label `.L118` which will later be replaced by the assembler/linker with an address. The format specifier `%e1` has been replaced with the register that was allocated to variable `i`. Finally, aiT extracts the annotation from the ELF executable, as shown in Listing 4.

Besides its use for the refinement of values, the annotation mechanism can also be used to insert "assert" annotations about known value ranges of variables or function parameters (see Listing 5). aiT will then report if any assertion is violated.

Listing 3 Assembly code generated by CompCert for the example in Listing 2.

```
.L116:
; int i = x;
  fctiwz   f13, f1
  stfdu    f13, -8(r1)
  lwz      r5, 4(r1)
  addi     r1, r1, 8

.L117:
; __builtin_ais_annot("try instruction %here { enter with: %e1 =
    ↪ 0..9; };", i);

.L118: .L119:
; return data[i];
  addi     r3, r1, 16
  rlwinm   r4, r5, 3, 0, 28 ; 0xfffffff8
  lfdx     f1, r3, r4

.L120:
  addi     r1, r1, 96

.L121:
  blr

...

.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; \};"
.ascii "\n"
```

Listing 4 Annotation extracted by aiT.

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

Listing 5 Assertion of input values of a function to be validated by aiT.

```
int func(int a, int b, int c)
{
    __builtin_ais_annot("try instruction %here {\n"
                        " assert always enter with: %e1 in (0..7);\n"
                        " assert always enter with: %e2 in (2..4);\n"
                        " assert always enter with: %e3 in (1..9);\n"
                        "};", a, b, c);
    ...
}
```

Listing 6 Specifying a bound for a data-dependent loop.

```
int strncmp_x(char s[], char t[], int len)
{
    int i;
    for (i = 0; i < len && ...; i++) {
        __builtin_ais_annot("try loop %here bound: %e1;", len);
        ...
    }
    return 0;
}
```

Listing 7 Providing unrolling hints for loops for improved precision in aiT.

```
#define MAX_STR_LEN 50

void strcpy_x(char s[], char t[])
{
    int i = 0;
    while (( s[i] = t[i] ) != '\0') {
        __builtin_ais_annot("try loop %here mapping { default unroll:
            ↪ %e1; }", MAX_STR_LEN);
        ...
    }
}
```

Loop and Recursion Bounds

Loop or recursion bounds cannot always be automatically derived by aiT's value analysis. A (probably overestimated) annotation can be specified in the source code of a common library routine to ensure that aiT can compute reasonable results without annotation effort or to increase analysis precision at specific code locations. If necessary, users of aiT can improve this annotation by giving more specific annotations for the actual analysis context. Listing 6 shows a data-dependent loop where the bound depends on the input parameter of the surrounding function. This fact can easily be expressed with `__builtin_ais_annot()`.

Sometimes the analysis precision can be greatly improved if the first x iterations of a loop can be distinguished. aiT supports this via virtual unrolling[1]. Note that this does not change the binary, nor does it affect the compilation process in any way. Instead, aiT uses analysis contexts to distinguish the first n loop iterations from all following ones. An annotation that enables the virtual unrolling of the first 49 iterations of a loop is shown in Listing 7.

In case of busy-waiting loops, no loop bound can be derived statically. However, it is also not easy to derive the maximal number of iterations manually, because this number depends on the execution time of the loop body. aiT supports a way to specify the loop bounds of busy-waiting loops depending on their worst-case waiting times, see Listing 8.

Finally, in automotive software, it is often the case that some implicit recursion happens during error handling. For these cases, we need to specify recursion bounds as shown in Listing 9.

[1] For historical reasons, aiT uses the name *virtual unrolling*, but *virtual loop peeling* might be a better name.

Listing 8 Specifying a time bound for a busy waiting loop.

```
void openCanSocket(volatile struct device_t* device)
{
    ...
    // Busy wait for ACK. Assume a worst-case timing of 23 us
    while(device->bus_data != 0x00) {
        __builtin_ais_annot("try loop %here takes: 23 us;");
    }
    ...
}
```

Listing 9 Specifying a recursion bound and an incarnation bound for a recursive routine.

```
void errorHook(unsigned char err_code)
{
    __builtin_ais_annot("try routine %e1 {\n"
                        "    recursion bound: 1;\n"
                        "    incarnation limit: 1;\n"
                        "}", &errorHook);
    ...
}
```

Memory Areas

The properties and contents of memory areas can also be specified for aiT. For example, special care needs to be taken when accessing memory-mapped sensors and other devices which provide data via asynchronously updated buffers. We can specify that these buffers are read-only and volatile, see Listing 10.

Another common pattern—shown in Listing 11—is to copy calibration data from ROM to RAM once when the system boots. Without further annotations aiT usually cannot know which data is stored in the calibration vector. With the `copy area` annotation, the precision of the analysis can be improved.

Listing 10 Memory areas that are used for external devices can be marked accordingly.

```
volatile char* device_buffer[128];

void init_device()
{
    __builtin_ais_annot("area %e1 width %e2 {\n"
                        "    readable: true;\n"
                        "    writable: false;\n"
                        "    volatile;\n"
                        "};", &device_buffer, sizeof(device_buffer));
    ...
}
```

Listing 11 A source level annotation to specify which data is copied from ROM into RAM.

```
volatile char calibration_data[__CALIBRATION_ROM_SIZE];

// setup at boot time
void init_calibration_data()
{
    __builtin_ais_annot("copy area %e1 width %e2 from %e3;}",
                        &calibration_data,
                        __CALIBRATION_ROM_SIZE,
                        (void *)__CALIBRATION_ROM_START);

    memcpy((void *)calibration_data, (void *)__CALIBRATION_ROM_START,
        ↪  __CALIBRATION_ROM_SIZE);
}
```

4 Interactions with Compiler Optimizations

Compiler optimizations can complicate the transmission of source-level annotations to the compiled code: if done carelessly, optimizations can remove annotations, or render them inapplicable to the code after optimization.

Preservation guarantees for annotations

As mentioned in section 2, CompCert's proof of semantic preservation guarantees that annotations are not erased during compilation, unless they occurred in parts of the code that are unreachable during execution, and that they not reordered or moved in the generated code, relative to each other and relative to other observable actions (such as external function calls and accesses to volatile variables). The proof does not rule out the possibility that optimizations would move annotations around pure, non-observable computations. However, this is not the case for CompCert's optimizations, which are conservative and make no attempts to optimize around calls to unknown functions, which include annotation statements. Another reason why CompCert preserves the position of annotations relative to the code is that it currently performs no loop optimizations, as discussed below.

Invariance of annotation texts

CompCert is agnostic with respect to the annotation language: it gives no specific meaning to the annotation strings and never modifies them during optimization. Consequently, an annotation that mentions functions or variables by their names can become pointless as a result of optimization.

Consider the example shown in Listing 12. After inlining, the annotation occurs within function `g` but still refers to an instruction in function `f`. The hardcoded reference to `f` in the annotation text must be replaced by a relative code position `%here`.

Similarly, if a static variable is mentioned by name in an annotation but unused anywhere else, CompCert will remove this variable and make the annotation meaningless. To avoid this risk, the variable should appear as an explicit argument of the annotation (see Listing 13).

Moreover, dead code elimination may remove annotations that have a global effect (see Listing 14).

Finally, some AIS annotations about function calls can become inapplicable if the call is

Listing 12 A source level annotation that is not robust regarding inlining.

```
static inline void f(void)
{
    __builtin_ais_annot("try routine 'f' ...");
    ...
}

int g(int x)
{
    ...
    f();
    ...
}
```

Listing 13 Explicit references to variables increase robustness.

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

turned into a jump by CompCert's tail call optimization. The workaround here is to turn tail call optimization off.

Towards loop optimizations

It is well known that program annotations that bound the number of iterations of a loop are difficult to maintain in the presence of loop optimizations [12]. CompCert does not address this problem since currently it does not perform any optimizations over loops. If classic loop optimizations were added in the future, they would combine poorly with loop count annotations expressed with `__builtin_ais_annot()`. First, most optimizations over loop nests, such as loop interchange or loop blocking, change the order in which iterations are performed. Hence, they do not apply if the loop body can perform observable operations such as I/O. CompCert's annotations being observable operations, the presence of one or several `__builtin_ais_annot()` to give loop bounds would inhibit these optimizations.

Second, optimizations such as loop unrolling make upper bounds on the number of loop iterations inaccurate (unrolling by a factor of k divides the number of iterations by k). Yet, in the CompCert approach, such an optimization is not allowed to rewrite the annotation to adjust the loop count, because this would change the observable behavior of the annotation according to the formal semantics. This is a real conundrum with no easy workaround.

Listing 14 Dead code elimination may remove annotations.

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

5 Related Work

CompCert [11, 3] already supports an annotation mechanism via `__builtin_annot()` [6]. There, the annotation string is printed as a comment in the generated assembly code. An additional tool can be used to parse these comments and to generate annotations, e.g. for aiT. Our work makes this extra annotation generator superfluous, as we print annotations that are (a) already in the right syntax to be understandable by aiT and (b) are directly stored in the executable binary.

The ENTRA (Whole-Systems ENergy TRAnsparency) Deliverable D2.1 "Common Assertion Language" [8] describe a similar mechanism to transfer properties from the source to the machine level. Pragmas are used to specify properties which are translated to comments written as inline assembler statements. These comments need to be extracted from the assembler files, as they are not stored in the final binary.

WCC [9] also uses pragmas to specify properties on the source code level. During compilation, which also contains steps to optimize the worst-case timing behavior, the compiler framework translates these properties into annotations for aiT in order to steer the WCET estimation of intermediate binaries. WCC only covers a subset of the annotations possible with AIS—loop bounds and linear flow constraints—whereas our approach allows to exploit the full power of the AIS annotation language. On the other hand, WCC is able to transform the annotations when applying optimizations like loop unrolling.

Li, Puaut and Rohou [12] present a framework in which annotations on the source code are transformed into annotations on the binary level in the presence of compiler optimizations. They focus on loop count annotations and their preservation through classic loop optimizations. Our approach cannot deal with loop optimizations yet, but supports a more general annotation language and provides formal correctness guarantees.

aiT allows to extract AIS annotations from source code via special markers in C comments [1], for example: `/* ai: loop here bound: 10; */`. However, the special program point `here` might not be resolvable due to compiler optimizations. Moreover, whenever source code annotations are extracted from a source file, the whole file is scanned for AIS comments independent from any `#if`, `#ifdef`, or `#ifndef` preprocessor directives.

TuBound [13] uses pragmas to annotate additional knowledge for the timing analysis. In contrast to aiT, which operates on fully linked binaries, TuBound incorporates a compiler and takes C code as input.

6 Conclusions and Future Work

CompCert's annotation mechanism via `__builtin_ais_annot()` enables programmers to *reliably* annotate flow facts on C source code level and reason about C variables instead of using code addresses or processor registers. Its versatility allows to exploit the full power of the AIS annotation language used by aiT. These annotations are automatically carried through the compilation chain and the linked executable into aiT without using external annotation files. Thus, version mismatch between executable and annotations is successfully prevented, which is especially useful for binary code libraries. Program points and other addresses survive recompilation, thus easing the maintenance effort needed for annotations.

There are two shortcomings of the current implementation of `__builtin_ais_annot()`: First, due to its treatment as a call to an external function, it cannot be placed at the top-level of a compilation unit, unlike, for example, a variable declaration. Second, since all annotations are merged in a single section, they cannot be extracted individually. We wish to address these shortcomings in future work.

References

1. AbsInt Angewandte Informatik GmbH. a^3 for PPC User Documentation (Version 18.04).
2. AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzer. `http://www.absint.com/ait/`.
3. AbsInt Angewandte Informatik GmbH. CompCert: formally verified optimizing C compiler. `http://www.absint.com/compcert/`.
4. AbsInt Angewandte Informatik GmbH. TimeWeaver: Hybrid Worst-Case Timing Analysis. `http://www.absint.com/timeweaver/`.
5. Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Trans. Embedded Comput. Syst.*, 13(4):82:1–82:37, 2014. `doi:10.1145/2560033`.
6. Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *ERTS 2012: Embedded Real Time Software and Systems*, 2012.
7. Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza (Burguière), Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingenieurs de l'Automobile*, 807:26–42, 2010.
8. ENTRA Consortium. Deliverable D2.1 "Common Assertion Language". `http://entraproject.ruc.dk/deliverables`.
9. Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, 46(2):251–300, 2010. `doi:10.1007/s11241-010-9101-x`.
10. Xavier Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43(4):363–446, 2009.
11. Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert – a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems*, 2016.
12. Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of flow information: Reconciling compiler optimizations and WCET estimation. In Mathieu Jan, Belgacem Ben Hedia, Joël Goossens, and Claire Maiza, editors, *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versaille, France, October 8-10, 2014*, page 97. ACM, 2014. `doi:10.1145/2659787.2659805`.
13. Adrian Prantl, Markus Schordan, and Jens Knoop. Tubound - A conceptually new tool for worst-case execution time analysis. In Raimund Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Prague, Czech Republic, July 1, 2008*, volume 8 of *OASICS*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.
14. Simon Wegener. Towards Multicore WCET Analysis. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis, WCET 2017, June 27, 2017, Dubrovnik, Croatia*, volume 57 of *OASICS*, pages 7:1–7:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. `doi:10.4230/OASIcs.WCET.2017.7`.
15. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3):36:1–36:53, 2008. `doi:10.1145/1347375.1347389`.