Verifying Timing Properties of Safety-Critical Embedded Software by Abstract Interpretation

C. Ferdinand, R. Heckmann
AbsInt Angewandte Informatik GmbH, Science Park 1, D-66123 Saarbrücken, Germany

Abstract: Many tasks in safety-critical embedded systems have hard real-time characteristics. A schedulability analysis has to be performed in order to guarantee that all timing constraints will be met. It requires the worst-case execution time (WCET) of each task in the system to be known prior to its execution. AbsInt’s worst-case execution time analyzer aiT computes safe and precise upper bounds for the WCETs of tasks taking into account cache and pipeline behavior. Information required for WCET estimation such as computed branch targets and loop bounds is determined by static analysis. For complex situations where aiT’s analysis methods do not succeed, a convenient specification and annotation language was developed. The analysis results are determined without the need to change the code and hold for all executions with arbitrary input.

Keywords: Safety, Timing Validation, Schedulability Analysis, WCET (worst-case execution time) Prediction

1. Introduction

Failure of a safety-critical application on an embedded processor can lead to severe damage or even loss of life. Also for non-safety-critical applications, software failure may necessitate expensive updates. Therefore, utmost carefulness and state-of-the-art machinery have to be applied to make sure that an application is working properly. To do so lies in the responsibility of the system designer(s).

Many tasks in safety-critical embedded systems have hard real-time characteristics. Real-time systems are typically composed of a set of tasks with specified deadlines (mostly dictated by the surrounding physical environment). Failure to meet deadlines may be as harmful as producing wrong output or failure to work at all. Therefore, a schedulability analysis (also called timing validation) has to be performed in order to guarantee that all timing constraints will be met [9]. It requires the worst-case execution time (WCET) of each task in the system to be known prior to its execution. Since this is not computable in general, estimates of the WCET have to be calculated. These estimates have to be safe, i.e., they must never underestimate the real execution time. Furthermore, they should be tight, i.e., the overestimation should be as small as possible.

Yet determining such good WCET estimates is a difficult problem because of the characteristics of modern software and hardware. Caches, branch target buffers, and pipelines are used in virtually all performance-oriented processors. Consequently the timing of the instructions depends on the execution history.

Hence, the widely used classical methods of predicting execution times are not generally applicable. Software monitoring and dual-loop benchmark change the code, what in turn changes the cache behavior. Hardware simulation, emulation, or direct measurement with logic analyzers cover only a tiny subset of the huge set of all possible executions. Moreover, if one takes into account the rate at which the size of typical avionics or automotive programs is increasing, maintaining the coverage of tests at the same level as today is increasingly expensive.

AbsInt’s worst-case execution time analyzer aiT solves these problems by a combination of abstract interpretation and integer linear programming. It reads two kinds of input: the executable program containing the tasks to be analyzed, and user input providing additional information. This information falls into two main classes: global information like the clock rate of the microprocessor and the access times of memory, and local information addressing specific program points. This local information complements the results of the various static analyses performed by aiT. In the sequel, we shall introduce these analyses and the respective possibilities for user information. Before doing so, we explain how users may refer to program points.

2. Program Points

User information may be provided in two ways: as specifications in a separate parameter file called
AIS file, or as special comments in the source code (see section 4). aiT is able to scan the source files for such comments, but it also works if no source is available; WCET analysis is based on the executable program.

Both kinds of user information need to refer to program points for various reasons, e.g., as points of reference or as branch targets.

- The simplest way to refer to a program point is by its absolute address in the executable. Yet this kind of program point specification is not very convenient. If the application is modified and then recompiled, the absolute addresses will change and need to be adapted in the user information. This is true even for unmodified routines since they may move to a different location in memory.

- These problems can be avoided to some extent by using relative addressing, e.g.,

  "main" + 0x20 bytes

  describes a program point by an address relative to the entry of routine main. If the application is modified and then recompiled, only the relative addresses in the modified routines must be adapted.

- Program points may also be specified by symbolic descriptions. For instance,

  - "prime" + 2 loops
    is the beginning of the second loop in routine prime.
  - "find" + 3 reads
    is the third instruction reading from memory in find, and
  - "watch" + 1 call
    is the first call instruction in watch.

  Further examples of symbolic descriptions will be presented in the next few sections.

- file 'Name' line Number
  refers to line Number in source file Name. The translation of this line into an address in the executable relies on the line information that is part of the executable (see section 4).

- The program point description here can only be used in source code annotations and then refers to the line where the annotation starts. This line information is translated to an executable address in the same way as the explicit line numbers presented above.

### 3. General Structure of aiT

aiT works on executables because the source code does not contain information on register usage and on instruction and data addresses. Such addresses are important for cache analysis and the timing of memory accesses in case there are several memory areas with different hardware realizations. There are aiT versions for several processor architectures: HCS12/STAR12, ARM7 TDMI, C16x/ST10, TMS320C33, and Motorola PowerPC MPC 555/565/755. They share a common structure [3] described in the following subsections (see Figure 1).

#### 3.1 Decoding and Control-Flow Reconstruction

In the first step a parser reads the executable and reconstructs the control flow [10]. This requires some knowledge about the underlying hardware, e.g., which instructions represent branches or calls. The reconstructed control flow is described as a combined call graph and control-flow graph, which serves as the input for micro-architecture analysis.

The decoder can find the target addresses of absolute and pc-relative calls and branches, but may have difficulties with target addresses computed from register contents. Thus, aiT uses specialized decoders that are adapted to certain code generators and/or compilers. They usually can recognize branches to a previously stored return address, and know the typical compiler-generated patterns of branches via switch tables. Yet non-trivial applications may still contain some computed calls and branches (in hand-written assembly code) that cannot be resolved by the decoder; these unresolved computed calls and branches are documented by appropriate messages and require user annotations. Such annotations may list the possible targets of computed calls and branches:

```
INSTRUCTION ProgramPoint
CALLS Target_1, ..., Target_n;
```
INSTRUCTION ProgramPoint
BRANCHES TO Target\textsubscript{1}, ..., Target\textsubscript{n};

ARM7 TDMI processors do not offer return instructions. Instead, various kinds of computed branches with the return address as target can be employed. aiT can recognize most of these branches as returns. The few remaining ones, mostly contained in library code, can be annotated as follows:

INSTRUCTION ProgramPoint
IS A RETURN;

The ProgramPoint, which refers to the computed call or branch instruction, and the Targets are points in the sense of section 2; they may be absolute or relative addresses or symbolic descriptions. A program point description particularly suited for CALLS and BRANCHES specifications is

"R" + n COMPUTED

which refers to the \textit{n}th computed call or branch in routine \textit{R}—counted statically in the sense of increasing addresses, not dynamically following the control flow. In a similar way, targets can be specified as absolute addresses, or relative to a routine entry in the form

"R" + n BYTES

or relative to the address of the conditional branch instruction, which is denoted by \textit{PC}.

Example 1: The library routine \textit{C_MEMCPY} in TI's standard library for the TMS470 consists of handwritten assembler code. It contains 2 computed branches whose targets can be specified as follows (keywords may be lower case or upper case):

\begin{verbatim}
instruction "C_MEMCPY" + 1 computed
branches to pc + 0x04 bytes,
   pc + 0x14 bytes,
   pc + 0x24 bytes;

instruction "C_MEMCPY" + 2 computed
branches to pc + 0x10 bytes,
   pc + 0x20 bytes;
\end{verbatim}

The advantage of such relative specifications is that they work no matter what the absolute address of \textit{C_MEMCPY} is.

If the application contains an array \textit{P} of function pointers, then a call \textit{P}[i](\textit{x}) may branch to any address contained in \textit{P}. aiT tries to obtain the list of these addresses automatically: If the array access and the computed call in the executable are part of a small code pattern as it is typically generated by the compiler, aiT notices that the computed call is performed via this array. If furthermore the array contents are defined in a data segment so that they are statically available, and the array is situated in a ROM area so that its contents cannot be modified, then aiT automatically considers the addresses in the array as possible targets of the computed call.

If array access and computed call are too far apart or realized in an untypical way, aiT cannot recognize that they belong together. Similar remarks apply to computed branches via switch tables. In both cases, the array or table belonging to the computed call or branch can be declared by the user. The declaration starts like the ones described above:

\begin{verbatim}
INSTRUCTION ProgramPoint
CALLS VIA ArrayDescriptor;
\end{verbatim}

\begin{verbatim}
INSTRUCTION ProgramPoint
BRANCHES VIA ArrayDescriptor;
\end{verbatim}

Here, the ArrayDescriptor describes the address and the format of the table that contains the call or branch targets. These targets are extracted from the table according to the given format rules.

3.2 Value Analysis

Value analysis computes safe lower and upper bounds for the values in the processor registers for every program point and execution context. In many cases, lower and upper bound are identical, i.e., value analysis can predict the exact value. Yet if it should happen that the precision of a value analysis result is not satisfactory, the user may increase it by providing better bounds or an exact value:

\begin{verbatim}
INSTRUCTION ProgramPoint
IS ENTERED WITH
   Register\textsubscript{1} = Range\textsubscript{1}, ..., Register\textsubscript{n} = Range\textsubscript{n};
\end{verbatim}

Registers are specified by name, and \textit{Range} denotes the range of possible values as a closed interval, i.e., by specifying the smallest possible value \textit{n} and the largest possible value \textit{N} in the form \textit{n} \ldots \textit{N} or \textit{FROM} \textit{n} \textit{TO} \textit{N}. If you know an exact value (\textit{n} \textit{=} \textit{N}), then you need only specify this value.

Example 2:

\begin{verbatim}
instruction 0x9110
   is entered with
      r3 = 0,
      r7 = 0x10..0x1F;

instruction "_prime"
   is entered with
      r2 = from 0 to 20;
\end{verbatim}

The first specification says that the instruction at address \textit{0x9110} is always entered with \textit{0} in \textit{r3} and \textit{0x10} \textless{} \textit{r7} \textless{} \textit{0x1F}. (The values may be entirely different after executing the instruction.) The second specification tells \textit{0} \textless{} \textit{r2} \textless{} \textit{20} when \textit{_prime} is entered (for all calls of \textit{_prime}).

The results of value analysis are used for various purposes listed in the following subsections. Each purpose comes with possibilities for specifications.
better suited for that purpose than the general register-value specifications presented above.

### 3.3 Loop Bounds

WCET analysis requires that upper bounds for the iteration numbers of all loops be known. \texttt{aiT} tries to determine the number of loop iterations by loop bound analysis, a combination of value analysis and pattern matching, which looks for typical loop patterns. In general, these loop patterns depend on the code generator and/or compiler used to generate the code that is being analyzed. There are special \texttt{aiT} versions adapted to various generators and compilers that are quite successful in finding loop bounds automatically. Sometimes they rely on the assumption that generated code is well-behaved. For instance, a common type of loops in generated code is linear search in a sorted array, e.g.,

\begin{verbatim}
  while (x > *(x_table++)) ... 
\end{verbatim}

Here, there is the risk that \(x\) is greater than all table values so that the loop continues examining values beyond the end of the table in an uncontrolled way. Yet \texttt{aiT} assumes that the code generator has avoided this error situation by an extra test before the loop or by putting the largest possible value at the end of the table. Then the number of executions of the loop header is bounded by the size of the table. To be on the safe side, \texttt{aiT} issues a message asking the user to verify that the assumption is valid.

Despite all sophistication built into loop bound analysis, there may be some loops that are too complicated for automatic analysis. Bounds for such loops must be provided by user annotations. A maximum iteration number of \(j\) is specified as follows:

\begin{verbatim}
  LOOP ProgramPoint Qualifier MAX j; 
\end{verbatim}

A \texttt{ProgramPoint} description particularly suited for this purpose is

\begin{verbatim}
  "R" + n LOOPS 
\end{verbatim}

which means the \(n\)th loop in routine \(R\) counted from 1. \texttt{Qualifier} is an optional information. It may be one of the following:

- \texttt{begin} indicates that the loop test is at the beginning of the loop, as for C’s \texttt{while}-loops.
- \texttt{end} indicates that the loop test is at the end of the loop, as for C’s \texttt{do-while}-loops.

If the qualifier is omitted, \texttt{aiT} assumes the worst case of the two possibilities, which is \texttt{begin} where the loop test is executed one more time. The \texttt{begin/end} information refers to the executable, not to the source code; the compiler may move the loop test from the beginning to the end, or vice versa.

**Example 3:**

\begin{verbatim}
  loop "_prime" + 1 loop end max 10; 
\end{verbatim}

specifies that the first loop in \_prime has the loop test at the end and is executed at most 10 times.

### 3.4 Addresses of Memory Accesses

Using the values of the registers, value analysis tries to determine the addresses of memory accesses. These addresses are important for an analysis of the data cache and for determining the duration of the memory accesses. Value analysis usually works so good that only a few indirect accesses cannot be determined exactly. Address ranges for these accesses may be provided by user annotations of the form

\begin{verbatim}
  INSTRUCTION ProgramPoint ACCESSES Range; 
\end{verbatim}

Useful \texttt{ProgramPoint} formats for such specifications are simple instruction addresses and symbolic descriptions of the forms

- \texttt{"R" + n READS}
- \texttt{"R" + n WRITES}

meaning the \(n\)th instruction in routine \(R\) reading from memory and the \(n\)th instruction writing to memory, respectively.

A \texttt{Range} may be a single position in memory, or a range specified by a start and an end position, or an array name meaning the memory area covered by that array (this requires that the debug information in the executable contains the start address and the end address or the length of the array). A position may be an absolute address in memory or an address relative to the beginning of an array (this requires that the debug information in the executable contains the start address of the array).

**Example 4:** Assume array \texttt{TAB} is mapped to memory area 0x8100–0x81FF, and the first read instruction in routine \texttt{main} has address 0x8500. Then the following specifications are equivalent:

\begin{verbatim}
  instruction 0x8500 <br> accesses 0x8100 .. 0x81FF; 
  instruction 0x8500 <br> accesses "TAB"; 
  instruction "main" + 1 read <br> accesses "TAB"; 
\end{verbatim}

The following specifications are also equivalent to each other, but different from the ones above because they refer to the start address of the array instead of the entire extent of the array:

\begin{verbatim}
  instruction 0x8500 <br> accesses 0x8100; 
  instruction 0x8500 <br> accesses "TAB" + 0 bytes; 
  instruction "main" + 1 read <br> accesses "TAB" + 0 bytes; 
\end{verbatim}
These specifications are valid no matter whether the read instruction accesses the byte 0x8100 or the word starting at byte 0x8100.

3.5 Evaluation of Conditions

If a condition always evaluates to true or always to false, certain program paths are never executed. Therefore, their execution time does not contribute to the overall WCET of the program, and need not be determined in the first place. A similar effect is obtained by user annotations specifying the values of conditions:

CONDITION ProgramPoint
IS ALWAYS TRUE;

CONDITION ProgramPoint
IS ALWAYS FALSE;

The ProgramPoint specified should be a conditional branch (not a compare instruction).

Example 5: The following annotation specifies that no division by zero occurs in the library routine __rt_udiv of the ARM compiler for ARM7:

CONDITION __rt_udiv + 0x4c bytes
IS ALWAYS FALSE;

Alternatively, a user may directly specify that a certain basic block is never executed:

SNIPPET ProgramPoint
IS NEVER EXECUTED;

where ProgramPoint refers to an arbitrary instruction in the block, e.g., by its address or by any other way presented in section 2.

3.6 Cache and Pipeline Analysis

Cache analysis classifies the accesses to main memory into cache hits, cache misses, or accesses that may be both [2]. Pipeline analysis models the pipeline behavior to determine execution times for sequential flows (basic blocks) of instructions [6]. The result is an execution time for each basic block in each distinguished execution context.

Cache and pipeline analysis cannot be influenced by local annotations. The cache layout can be described by global specifications. For experimental purposes, there are also global specifications declaring that all memory accesses should be considered as cache hits, or as misses.

3.7 Path Analysis

Finally path analysis determines a worst-case execution path of the program from the timing information for the basic blocks. The program’s control flow is modeled by an integer linear program [11] so that the solution to the objective function is the predicted worst-case execution time for the input program.

Path analysis takes into account user-given flow facts that consist of linear constraints for the execution counts of several program points. For instance,

flow (0x100) + (0x200) <= 4 (0x300);

means that the number of executions of the block starting at address 0x100 plus the number of executions of the block starting at 0x200 is at most 4 times the number of executions of the block starting at 0x300. As always, relative addresses or symbolic program point descriptions may be used instead of these absolute addresses.

4. Source Code Annotations

User information can be written into a separate parameter file, or inserted into C source code files as special comments marked by the key string ai:

/* ai: specification1; ... specificationn; */

The names of the source files are extracted from the debug information in the executable.

Source code annotations admit a special program point or target here, which roughly denotes the place where the annotation occurs (due to compiler optimizations the debug information is not always precise). More exactly, aiT extracts the correspondence between source lines and code addresses from the executable. A here occurring in source line n then points to the first instruction associated with a line number ≥ n. Since the line information in the executable is created by the compiler, it becomes invalid when lines are added or deleted in the source file. Therefore the application must be recompiled whenever lines are added while annotating.

For loop annotations, it is not required that here exactly denotes the loop start address. It suffices that it resolves to an address anywhere in the loop as in the following example:

for (i=3; i*i <= n; i += 2) {
  /* ai: loop here end max 10; */
  ...
}

5. Other Annotations

Apart from the annotations described so far, many other properties can be declared in parameter or source files.

• To get any WCET results at all, you must specify upper bounds for the recursion depths of all recursive routines. These specifications are similar to the loop bound specifications described in section 3.

• aiT can be informed about the clock rate of the microprocessor: Knowing the clock rate, aiT can
display its results in real time units such as milliseconds. Without this information, all results are displayed in processor cycles.

- End specifications instruct aiT to stop reading the executable at a certain program point. A possible application is for instance to inform aiT that an interrupt routine called by a software interrupt does not return.
- You may specify that a memory area is read-only or write-only, contains data or code. You may also specify which data it contains.
- You may exclude certain routines from WCET analysis and supply their WCET directly.
- You may specify that a routine never returns (like `exit`).
- You may exclude certain routines from analysis and specify their WCET by hand.
- Program points can be given symbolic names for later reference.

6. Related Work

In contrast to most approaches proposed in the literature [7, 5, 4, 1], our annotations may refer to the source code, but do not extend the source language (annotations are comments), nor do they require a special compiler. Instead, aiT can analyze code generated by standard compilers. The correspondence between source code annotations and low-level object code is exclusively based on the debug information of the executable. Other than the annotations proposed elsewhere, ours cover the full spectrum between reference to source code lines (here) over symbolic descriptions (R + 1 loop) till routine-relative or absolute addresses, the latter being useful for annotating optimized code with instructions that cannot be attributed to a particular piece of source code.

The annotation languages proposed in [7, 5, 4, 1] are generally restricted to loop bounds and flow constraints, while ours are more general in that they also admit the specification of targets of computed calls and branches, register values, and addresses of memory accesses. On the other hand, the specialized flow languages, in particular the one proposed in [1], are more expressive and powerful than our flow constraints. Extensions in this direction are intended, but not yet realized to get a working system as soon as possible.

7. Conclusion

aiT allows to inspect the timing behavior of (time-critical parts of) program tasks. The analysis results are determined without the need to change the code and hold for all executions with arbitrary input.

aiT is a WCET tool for industrial usage. It has been evaluated by Airbus on safety-critical avionics programs with encouraging results [12, 8]. Information required for WCET estimation such as computed branch targets and loop bounds is determined by static analysis. For situations where aiT’s analysis methods do not succeed, a convenient specification and annotation language was developed in close cooperation with AbsInt’s customers. This effort has contributed to the good acceptance aiT has found among producers of real-time software.

aiT enables development of complex hard real-time systems on state-of-the-art hardware, increases safety, and saves development time. At a stage when the software is already available, but working hardware is not, the tool can be used for a performance evaluation. Based on the contributions of the program parts to the WCET one can make design decisions, e.g., with respect to static scheduling or code/data placement. The effects on the cache and pipeline can be viewed using the visualization options of the tool. Causes for unexpected local timing behavior can be identified in this way.

8. References


