# Evaluation of Automatic Flow Analysis for WCET Calculation on Industrial Real-Time System Code

Dani Barkah[†], Andreas Ermedahl[⋆], Jan Gustafsson[⋆], Björn Lisper[⋆], and Christer Sandberg[⋆]
[†]Volvo Construction Equipment AB, Component Division, Dept 14700, SE-631 85 Eskilstuna, Sweden
[⋆]Department of Computer Science and Electronics, Mälardalen University, Box 883, S-721 23 Västerås, Sweden
dani.barkah@volvo.com {andreas.ermedahl,jan.gustafsson,bjorn.lisper,christer.sandberg}@mdh.se

## Abstract

*A static Worst-Case Execution Time (WCET) analysis derives upper bounds for the execution times of programs. Such analysis requires information about the possible program flows. The current practice is to provide this information manually, which can be laborious and error-prone. An alternative is to derive this information through an automated flow analysis.*

*In this article, we present a case study where an automatic flow analysis method was tested on industrial real-time system code. The same code was the subject of an earlier WCET case study, where it was analysed using manual annotations for the flow information. The purpose of the current study was to see to which extent the same flow information could be found automatically. The results show that for the most part this is indeed possible, and we could derive comparable WCET estimates using the automatically generated flow information. In addition, valuable insights were gained on what is needed to make flow analysis methods work on real production code.*

## 1 Introduction

The *worst-case execution time* (WCET) is an important parameter when verifying real-time properties. A *static WCET analysis* estimates the WCET of a program from mathematical models of the hard- and software involved. If the models are correct, the analysis will derive a timing estimate that is *safe*, i.e., greater than or equal to the WCET. To be useful, the estimate must also be *tight*, i.e., provide little or no overestimation compared to the WCET.

To derive a timing bound for a program, information on both the *hardware timing characteristics*, as well as the program's *possible execution flows*, needs to be derived. The latter includes iteration bounds on loops, feasible paths through the program, etc.

The goal of *flow analysis* is to calculate such *flow information* as automatically as possible. Flow analysis research has mostly focused on *loop bound* analysis [12, 17, 21, 23, 24, 27], since these must be known in order to derive finite WCET estimates. Flow analysis can also identify *infeasible paths*, i.e., paths possible to take according to the program control-flow graph (CFG), but not actually feasible when executing the program. Infeasible path information is not required to find a WCET estimate, but may enable tighter WCET estimates, i.e., if the worst-case path is found to be infeasible. Recent case studies [31, 37] have shown that such information can help obtaining tighter WCET bounds on industrial code. This has stimulated the development of flow analyses calculating infeasible path information automatically [20, 23].

Commercial WCET analysis tools such as aiT [2], Bound-T [40], and RapiTime [32] are available today. However, obtaining safe and tight WCET estimates with these tools is still a laborious process, as has been verified in several case studies on industrial codes [7, 8, 14, 36, 37]. A major problem is typically that much information must be provided manually. In particular, the tools tend to require external information about possible program flows, which can be hard to manually provide since it typically requires a deep understanding of how the code works. Obviously, it should be interesting to test recently developed flow analysis methods on real production codes in order to see whether they can indeed alleviate this problem in practice, and to identify possible problems.

In this article we report on a case study [5], where a flow analysis method [23] was tested on code used in articulated haulers manufactured by Volvo Construc-

This research has been supported by the KK-foundation through grant 2005/0271. Travel support has been provided by the ARTIST2 European Network of Excellence.

tion Equipment [41]. The code is written in a way that gives many infeasible paths: thus, it is apt for evaluating analyses detecting such paths.

The code was subject to an earlier case study [37], where it was analyzed with the aiT tool [2] using manual annotations to constrain the program flow. Producing these annotations took several weeks: obviously, automatic methods for deriving this information would be highly desirable. The current study is a direct repetition of the previous one, using the aforementioned flow analysis method to derive program flow constraints. This made it possible to make a direct comparison between the two studies.

The contributions of this article are:

- We evaluate the flow analysis method of [23] on industrial code, measuring how many loops it can bound and how many infeasible paths it can find.
- We make a direct comparison of the WCET estimates calculated using automatically derived flow information, and using manually derived flow information from [37] respectively.
- We identify some problems arising when analyzing real production code, as opposed to academic benchmarks, and indicate how they can be solved.

The rest of this article is organized as follows: In Section 2, we give a brief introduction to WCET analysis and related work in the area. Section 3 briefly describes the used flow analysis method. Section 4 describes the analysed code and the system where it is run. Section 5 presents results from a previous study. Section 6 and Section 7 introduce the used WCET analysis tools and the experimental setup. Section 8 presents observations made together with new analysis features developed. In Section 9 we give the results, and discuss them. Finally, in Section 10, we draw some conclusions and give directions for further research.

## 2 WCET Analysis Overview and Related Work

Static WCET analysis is usually divided into three phases: a *flow analysis*, a *low-level analysis* where the execution times for sequences of instructions are decided from a performance model for the hardware, and a final *calculation* phase where the flow and timing information is combined to yield a WCET estimate. Some WCET analysis methods are *input-sensitive*, taking constraints on input values into account to produce tighter WCET estimates.

As mentioned, flow analysis derives information about the possible execution paths through the program. A *loop bound analysis* finds upper bounds to the number of iterations of loops. Several approaches exist: syntactically oriented, detecting certain loop patterns [24, 38], or semantically oriented, using Presburger arithmetics [27], a combination of data flow analysis and value analysis [12], or abstract interpretation and program slicing [17, 21, 23].

An *infeasible path analysis* finds infeasible paths in the program. Consider this example:

```
if (x<0) A else B; if (x>2) C else D;
```

Here, the true-branches for the `if` statements are in conflict[1], and the corresponding path **A-C** is infeasible. Restrictions on input data values may yield more infeasible paths: for instance, if `x>5`, then also the paths **A-D**, and **B-D** are infeasible. This can be detected by an input-sensitive analysis.

Infeasible path analysis has been developed by combining path enumeration, path pruning, and symbolic evaluation [4], by using value-dependent constraints [25], through partially-known variables [3], by identifying conflicts between variable assignments and branch conditions [9], by abstract execution [23], and by using Presburger arithmetics [20].

In low-level analysis, researchers have studied effects of various hardware enhancing features, like caches, branch predictors and pipelines [13, 29, 38]. A frequently used calculation method is IPET (Implicit Path Enumeration Technique), using arithmetical constraints to model the program structure, the program flow and low-level execution times [15, 27, 38].

Studies of WCET analysis of industrial code are not common. There are some reports on the use of commercial WCET tools for analyzing codes for space applications [27, 26, 33], and in avionics industry [18, 39]. The experiences from some case studies are compiled in [16]. These studies include WCET analysis of the OSE operating system [8, 36], code controlling welding equipment [14], and communication software in cars [7]. The aforementioned study of transmission code [37] should also be mentioned, as well as a study of WCET analysis of automotive software [31]. However, this is the first paper, to our knowledge, that evaluates automatic flow analysis for WCET calculation on industrial real-time system code.

## 3 Abstract Execution

The flow analysis method used in the presented case study, called *abstract execution* (AE), is briefly described in this section. For details, we refer to [23].

AE is a form of symbolic execution [21]. It is based on *abstract interpretation* (AI) [10], a framework for program analysis where elements of an *abstract domain*

[1] We assume that `x` is not updated in **A** and **B**.

**Figure 2. Motor graders, wheel loaders, excavators and articulated haulers developed by Volvo CE**

```
i = INPUT; // i = [1..4]
while (i < 10) {
   // point p
   ...
   i=i+2;
}
// point q
```

**(a) Example**

| iter | i at p |
|---|---|
| 1 | [1..4] |
| 2 | [3..6] |
| 3 | [5..8] |
| 4 | [7..9] |
| 5 | [9..9] |
| 6 | impossible |

**(b) Analysis**

**min.**
**#iter:** 3

**max.**
**#iter:** 5

**(c) Result**

**Figure 1. Example of abstract execution**

represent sets of values, or program states. For instance, sets of numerical values can be represented by intervals, and program states by mappings from addresses to intervals. AE executes the program in the abstract domain, with abstract values, and abstract versions of functions and operators. If the abstract domain uses intervals, then each numeric variable will hold an interval rather than a number, and each assignment will calculate a new interval from the current intervals held by the variables. The abstract value held by a variable, at some point, represents a set containing the actual variable values at that point. AE uses *abstract states*, with a program counter, and an *abstract store* representing the contents of the memory. AE is naturally input-sensitive since it is possible to constrain the values of input variables in the initial abstract store. Figure 1 illustrates how AE works for a loop, by iterating, and counting the number of iterations, until the abstract version of the loop condition surely returns `false` for the back edge.

Sometimes, abstract execution of a condition node will yield possible execution paths for both the `true`- and `false`-branch. In Figure 1 this occurs before iteration 4 and 5, where `i = [7..10]` and `[9..11]`, respectively. An abstract state will then be created for each outcome of the test. This means that there may be many simultaneous abstract states, representing different possible execution paths. The number of possible abstract states may grow exponentially with the length of these paths: thus, any algorithm for abstract execution must be able to *merge* states, which is typically done at program points where different paths join. If the states are merged using the least upper bound operator on the domain of abstract states, then the result is an abstract state safely representing all possible concrete states, but possibly with some loss of precision. Different strategies for merging will thus yield different tradeoffs beween analysis time and precision.

We have designed and implemented an algorithm for AE [23], which can generate different kinds of program flow constraints. It is a quite straightforward worklist algorithm, which iterates over a set of abstract states, generating new abstract states from old ones. Certain program points are taken to be *merge points* (typically where different paths join): the algorithm keeps a list of abstract states having reached a merge point, and merges new states arriving to a merge point with previous states at the same point. States in the merge list are eventually released, according to some rule. At present, the merged states are released in random order. The algorithm terminates when all abstract states have reached the final program point.

Flow constraints are typically generated for repeating program constructs, like loops. Our AE algorithm extends abstract states with *recorders*, which collect program flow information. Program parts to be analyzed have *collectors*, which successively accumulate this information from the states. Supported analyses are (upper and lower) loop bounds analysis, and several infeasible path analyses: *infeasible nodes*, which are nodes never executed in a certain context, *maximal node counts*, generalising the infeasible node analysis, *excluding node pairs*, pairs of nodes which can never be executed together, and *infeasible paths* longer than two nodes.

## 4 The Analyzed System

The code under study is used in articulated haulers and other vehicles manufactured by Volvo Construction Equipment (Volvo CE) [41]. Volvo CE is a world leading manufacturer of such equipment, see Figure 2.

The vehicles are controlled by a distributed control system, consisting of a set of networked ECU's equipped with the Infineon C167CS processor [28].

**Rubus Real-Time Operating System.** The software uses the Rubus real-time operating system [34]. Rubus is task-oriented, with three kinds of tasks. *Green* tasks are interrupt tasks. They have the highest priority in the system and preempt other tasks when released. *Red* tasks are hard real-time tasks triggered by the system clock: they are scheduled offline. Possible attributes for the red tasks are release time, period time, deadline, precedence and WCET. *Blue* tasks, finally, are event-triggered soft tasks, and are scheduled online at lower priority than red and green tasks.

Rubus can measure the execution time of tasks. This can be used for high-water-marking of the execution time, which then bounds the WCET from below. The current practice is to set the WCET attribute for red tasks by adding a safety margin to the measured WCET estimate.

The Rubus VS is built upon the Rubus Component Model (Rubus CM) which uses sets sets of interconnected Software Components (SC). Each SC has a set of input ports where data is received, and a set of output ports where data is produced.

**Task Code Properties.** The analyzed code consists of 13 red real-time tasks in the Transmission ECU for articulated haulers. The source code for the tasks is mainly written in C (some tasks include some assembler), and it is compiled with the Tasking compiler. All tasks have a simple code structure, with if-statements, just a few loops, only one nested loop, no recursion, and no dynamic calls or memory allocations. The larger tasks contain a large number of function calls, and some functions are used many times in many different contexts. The code is written in a style with many mutually exclusive conditions, which gives rise to many infeasible paths. For details, see Section 9 and [37].

**Single-shot Tasks.** Red Rubus tasks have *single-shot task semantics*[2]. Such tasks terminate after each invocation, and the OS is responsible for scheduling and (re)activating the task. Such tasks should not contain delay calls, or infinite loops waiting for invocations. Figure 3 shows the difference between single-shot- and traditional task coding.

Red tasks do not preempt each other in the Volvo CE system. At each invocation a single-shot task will be passed all its needed data, either through input parameters or globals, instead of keeping it on the stack. If a single-shot task cannot be preempted, then its effect on the environment can be analyzed by considering its code in isolation. We utilize this in the *SVVA analysis* described in Section 8.

[2]Sometimes called *one-shot tasks* or *run-to-completion tasks*.

```
taskEntryFunc(params) {
  // Read params
  // Do some work
  // Set params
  return;
}
```

**(a) Single-shot task**

```
taskEntryFunc( ) {
  while(1) {
    // Read locals & globals
    // Do some work
    // Set locals & globals
    wait_for_invocation();
  }
}
```

**(b) Traditional task**

**Figure 3. Different task models**

Furthermore, the single-shot task model makes it more easy to determine each task's input parameters, as well as the task's start and end point (the latter giving the code for which a WCET estimate should be derived), see Figure 3(a). For traditionally coded tasks, this is more difficult: see Figure 3(b).

In Rubus each red task has a dedicated C entry function, taking a single pointer as parameter. The pointer points to a global C struct which belongs to the task. The struct has three kinds of elements:

- *input* - pointers to data areas, which contain input data to the task (written by other tasks)
- *output* - pointers to data areas where the task can write its produced output (can only be read by other tasks), and
- *state* - pointers to data areas where persistent private data are stored between task invocations.

If there are no green tasks in the system, then the output and new state of a red task will be solely decided by its input and previous state, as if the code had executed in isolation.

## 5 The Previous Case Study

In the previous case study [37], the same 13 red tasks were analyzed with the aiT tool using only manually provided annotations. Providing the annotations took about seven weeks: this included annotations for bounding the loops, and specifying infeasible paths. Especially the latter proved to be laborious. Providing the infeasible path information gave up to 31% tighter WCET estimates compared to a minimum effort analysis using loop bounds information only. Compared with measured (unsafe) WCET estimates, the statically calculated WCET estimates (with infeasible path information) were in the range 4 – 33% higher. The results show that reasonably tight WCET estimates can be achieved for this kind of code, using a static WCET analysis tool, and that infeasible path information can give substantially tighter WCET estimates, however at a potentially high cost in labor time.

## 6 The SWEET WCET Analysis Tool

SWEET [15, 22] is a WCET analysis research prototype tool [30]. SWEET can handle full ANSI-C programs including pointers, unstructured code, and recursion. The basic analysis steps of SWEET are shown in Figure 4. This case study only used the flow analysis of SWEET, so we only describe this analysis here. The low level and calculation parts of SWEET were replaced by aiT in this case study, since SWEET does not support the C167CS processor.

The flow analysis of SWEET is automatic and input-sensitive. Unlike most WCET analysis tools, SWEET is integrated with a compiler and performs the flow analysis on its intermediate representation NIC [44].

Constraints on input values are given in SWEET's annotation language. Numeric variables are constrained by intervals. Pointer constraints are sets of abstract addresses, each representing a range of NIC addresses. Annotations can constrain the variable values in specific program points. Normally, when constraining inputs, this is the program entry point.

SWEET uses abstract execution as described in Section 3. SWEET also includes a traditional *value analysis* (VA) by abstract interpretation [10], which gives a safe, potentially pessimistic, estimation of the possible sets of states in different program points. Currently supported abstract domains in the AE and VA are intervals [22], congruences [6], and their product domain.

SWEET performs a *program slicing* [42] in order to restrict the AE and VA to only those program parts and variables that may affect the program flow [35]. The program slicing can also be used to see what parts of a code might be affected by which input variables.

For large programs both the AE and the VA might need to allocate many abstract states. Each state can be large since each variable or aggregate data structure not removed by the slicing, should be assigned a corresponding abstract value. To minimize the memory needs a *copy-on-write* approach is therefore used [11], allowing states to *share* variable or aggregate data structure assignments with identical abstract values.

SWEET has a rich number of options allowing the user to tailor the analysis, and to control the tradeoff between precision and analysis time. It allows explicit control over merge point placement, different types of infeasible path detection can be done at several levels of precision, different underlying abstract domains can be used, sharing can be turned off or on, etc.

The output from SWEET's different flow analyses is a set of *flow facts*, which basically are sets of linear inequalities constraining the possible values of execution counters. The flow fact concept is described in

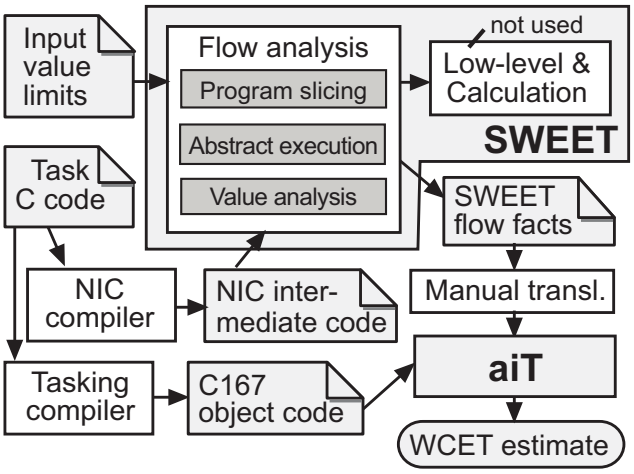


**Figure 4. The experimental setup**

e.g., [15]. Flow facts are given relative to *scopes*, which typically correspond to loops and functions. The flow facts produced by SWEET are *context sensitive*, where the contexts are call strings of scopes. For example, a function can obtain different flow facts at different call sites, which may yield more precise flow facts and a tighter WCET.

## 7 The Experimental Setup

The setup was done to mimic the earlier case study [37], using the aiT tool to do the final WCET calculation. The manually provided aiT annotations in [37] were replaced by annotations which were manually translated from SWEET's automatically derived flow facts. See Figure 4 for an illustration.

aiT is a commercial WCET analysis tool from AbsInt GmbH [2]. It analyses executable binaries, and supports the Infineon C167CS architecture among others [43]. Like SWEET, aiT has an annotation language for providing external information to the analysis [19]. Some important annotations to constrain the possible program flow are: *loop bounds*, *maximal recursion depth*, *dead code*, *outcome of conditions*, and *possible values of registers*. The aiT annotations are *context-insensitive*, meaning that an annotation within a function must be valid irrespective of the context from where the function is called.

The translation process from SWEET flow facts to aiT annotations was quite straight-forward. The flow facts for infeasible paths could be expressed using aiT annotations for flows and conditions. However, we had some minor problems; one was that not all the flow facts could be translated, since SWEET's flow facts are context sensitive whereas aiT's annotations are not. Another one was that in some cases the code generated by the Tasking compiler had a slightly different CFG

```
Task(INPUT_T *in) {
   static int prev;
   int diff,i;
   diff = in->cur - prev;
   for (i=0; i<diff; i++)
      { /* do some work */ }
   prev = in->cur;
}
```

**Figure 5. Task code using a state variable**

than the corresponding NIC code. Thus, we had to carefully compare the generated codes to make sure that the NIC basic blocks, referred to by SWEET flow facts, had a counterpart in the C167CS binary code.

## 8 Observations and New Features

During the case study we made some observations which sometimes inspired us to develop new analysis features. The major observations, and the new features are presented below.

**Input-value dependent conditions.** Many conditions in the code depended on task inputs. This provides an argument for input-sensitive flow analyses, since they can potentially produce better flow information for such codes. In addition, the conditions in a task often depended on *many* task inputs (see Section 9 for details). Thus the program slicing was, for many tasks, not able to remove many inputs. This resulted in large abstract states in the analysis.

**Large and complex input sets.** Input sets to different tasks were often very large and consisted of several complex aggregate data structures. We believe that this common for industrial codes. This added further to the size of the abstract states, and also made the task of constraining inputs more laborious. To ease the latter we developed several new features in SWEET's value annotation language, including better support for constraining the values of aggregate data structures and pointers.

Due to the large input sizes, and since input data structures often resided in various C files with types hidden by typedefs, it became problematic to give annotations for some globals. Similar to many assembly languages, NIC's data has no declared types. Thus, it is possible to assign any type to a NIC variable. We therefore had to be very careful when giving input value annotations.

To help detecting erroneous annotations, we implemented an *annotation type checker*, which generates a warning if the type of a given annotation does not correspond to the type assumed by the AE for the initial value set by the NIC compiler. This feature turned out to be quite helpful.

**Algorithm 1:** The SVVA analysis.

SVVA($T, S_{init}$)
**Input:** $T$: A task to find state variables for
$S_{init}$: An initial state
1. $S_{new} := S_{init}$
2. **repeat**
3. $S_{prev} := S_{new}$
4. $S_{new} := \text{VA}(T, S_{prev}) \sqcup S_{prev}$
5. **until** $S_{prev} = S_{new}$
6. PrintStateVariableAnnots($S_{new}$)

An erroneous or missing annotation can also result in a run-time error in the AE. The frequency of such errors prompted us to improve the AE error messages with better information to trace the error's origins.

**Globals shared between tasks.** Global data in C is either initialized by the programmer or given a default zero (NULL) value. Inputs shared by several different tasks are, in the Volvo CE system, implemented as global C data structures. Thus, it was in many cases not safe to use the default values as input value constraint since the globals might be assigned another values by other tasks. We therefore had to identify such globals and in many cases manually constrain their values. Careful inspection of the code, often in collaboration with the Volvo CE engineers, had to be made to ensure the correctness of the provided input value constraints. We believe that this situation is similar for many real-time systems where tasks communicate through global shared variables.

**Unknown values of state variables.** As described in Section 4, each analyzed task uses three different data areas for communicating results between tasks (*input* and *output*) and to hold values between its invocations (*state*). Both input and state variables may be used to decide the outcome of conditionals and must be considered by the flow analysis. For the analyzed system, the values of inputs were well known and could be annotated. However, the possible values of the state variables were, in general, not known. In most cases it was also hard to find these values by code inspection.

We believe that such state variables are common in many task-oriented systems. Figure 5 illustrates the principle. Here, the state variable `prev` (declared `static`, thus persistent) retains its value until the next invocation of the task.

**State Variable Value Analysis.** To derive constraints for the values of state variables automatically we developed an analysis, called *State Variable Value Analysis (SVVA)*. SVVA is performed as a fixpoint calculation over a task $T$, as shown in Algorithm 1. $S_{init}$ is the initial state at the start of the first invocation of $T$, with annotations of input variables. VA

| Task | Source | LOC | Funcs | Loops | Conds |
|---|---|---|---|---|---|
| 1 | 3.6 | 55 | 1 | 0 | 4 |
| 2 | 4.5 | 56 | 1 | 0 | 4 |
| 3 | 4.7 | 58 | 3 | 0 | 4 |
| 4 | 5.2 | 72 | 1 | 0 | 13 |
| 5 | 6.3 | 86 | 2 | 0 | 9 |
| 6 | 4.9 | 43 | 5 | 1 | 10 |
| 7 | 9.3 | 123 | 4 | 1 | 32 |
| 8 | 8.2 | 119 | 5 | 0 | 17 |
| 9 | 5.5 | 49 | 5 | 0 | 9 |
| 10 | 10.9 | 195 | 3 | 0 | 40 |
| 11 | 8.8 | 188 | 5 | 0 | 43 |
| 12 | 40.7 | 707 | 20 | 0 | 165 |
| 13 | 565.0 | 12765 | 97 | 18 | 2737 |

**Table 1. Properties of the 13 analyzed tasks**

| Task | Inpt | ISize | FInpt | ReqAn | DerAn |
|---|---|---|---|---|---|
| 1 | 6 | 31777 | 6 | 4 | 0 |
| 2 | 6 | 82 | 6 | 6 | 0 |
| 3 | 7 | 31812 | 7 | 7 | 0 |
| 4 | 13 | 31842 | 13 | 12 | 2 |
| 5 | 9 | 70 | 9 | 7 | 0 |
| 6 | 7 | 78 | 7 | 7 | 0 |
| 7 | 11 | 132 | 11 | 18 | 3 |
| 8 | 9 | 115 | 5 | 17 | 11 |
| 9 | 8 | 81 | 8 | 8 | 4 |
| 10 | 20 | 31899 | 20 | 17 | 8 |
| 11 | 24 | 176 | 24 | 16 | 3 |
| 12 | 58 | 32245 | 55 | 25 | 15 |
| 13 | 1044 | 50863 | 250 | 69 | - |

**Table 2. Input dependencies and annotations**

stands for "value analysis", and can be any analysis that, given an initial abstract state and a task as input, returns a final abstract state which holds a safe over-approximation of all variable values. In our evaluations we used SWEET's value analysis for this purpose, but the AE should work as well. The fixpoint is an abstract state with possible ranges for the state variable values, from which annotations for the state variables are generated.

In order to reach a fixpoint faster, knowledge about state variable values can be provided through annotations to the analysis. E.g., in our case study several tasks had a counter whose value range was obvious. The corresponding state variables will then be excluded from the SVVA analysis. In order to make the SVVA analysis even more efficient, we can (optionally) perform it only on the program slice with respect to the state variables at the exit point of the task.

## 9 Experimental Evaluation

Table 1 gives some code properties for the 13 analyzed tasks. **Source** gives the source code size in kilobytes (kb). **LOC** gives the number of C-code lines with comments and blank lines removed. **Funcs** gives the number of functions reachable within the task and **Loops** and **Conds** gives the total number of loops and conditions (including loop-exit conditions) within these functions. Tasks 1 – 11 are quite small, whereas task 12 and especially task 13 are substantially larger. Only a few of the tasks contain loops. The loop in task 7 was written in assembler, which we had to remove to make the source compatible with the NIC compiler. We rewrote the part of the code which depended on the loop. All tasks contain conditions, in particular task 13.

Table 2 gives some input-dependency and annotation measures for the 13 analyzed tasks. **Inpt** gives the number of global variables and aggregate data structures which are inputs to the given task. **ISize** gives the total size in bytes allocated for the inputs by the NIC compiler. **FInpt** gives the number of inputs which may affect the program flow. These are the inputs kept after making a program slicing upon all conditions in the code. Since our program slicing treats each aggregate data structure as one single data object, such a data structure will be kept as long as a one of its members fields holds a value or points to something which might affect the program flow.

There are several programs with really large input sizes, even though the number of input data structures might be small. Thus, many of the tasks use large input data structures. For most programs very few inputs can be sliced away: thus, there are many remaining variables and aggregate data structures which may affect the program flow.

The number of input value annotations required by the AE to produce flow facts is given by **ReqAn**. This number does not always correspond to the number of globals which may affect the program flow. This is because different fields in an aggregate data structure might need to be separately annotated and since some inputs, e.g., global pointers, might be given an initial value by the compiler which never is updated by the program. **DerAn** holds the number of state variable input value annotations derived by SVVA. The value 0 in the column means that no state variables were used, the "-" (for task 13) indicates that the value could not be calculated (see next page).

Table 3 gives some measures on the loop bound and infeasible path analyses performed. Column **BL** gives the number of bounded loops by the AE loop bound analysis. The value 0 means that the analysed program contains no C-code loops (recall that the assembler loop in task 7 was removed). We conclude that SWEET is able to automatically derive loop bounds for all C-code loops. The analysis was preceeded by a program slicing on loop exit conditions only. Thus, more code and inputs could be sliced away compared to the infeasible path analysis, resulting in smaller abstract

| Task | BL | Time | Cov | IN | EP | IP | MC | DA |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 2.75 | 100% | 0 | 0 | 0 | 10 | 0 |
| 2 | 0 | 0.02 | 100% | 0 | 0 | 0 | 10 | 0 |
| 3 | 0 | 2.72 | 100% | 0 | 0 | 0 | 10 | 0 |
| 4 | 0 | 7.47 | 100% | 0 | 4 | 0 | 31 | 4 |
| 5 | 0 | 0.12 | 100% | 0 | 2 | 1 | 25 | 3 |
| 6 | 1 | 0.07 | 92% | 2 | 0 | 0 | 21 | 2 |
| 7 | 0 | 5.65 | 100% | 0 | 7 | 5 | 73 | 12 |
| 8 | 0 | 0.41 | 100% | 0 | 5 | 0 | 45 | 5 |
| 9 | 0 | 0.04 | 100% | 0 | 0 | 0 | 23 | 0 |
| 10 | 0 | 15.08 | 97% | 9 | - | - | 91 | 2 |
| 11 | 0 | 4.35 | 100% | 0 | 4 | 0 | 91 | 4 |
| 12 | 0 | 173.34 | 99% | 354 | 10 | 8 | 1067 | 19 |
| 13 | 18 | - | - | - | - | - | - | - |

**Table 3. Analysis and flow information details**

states in the AE and less remaining code to analyse. A manual inspection of the analyzed loops gave that only one of the analyzed loops had an iteration count which depended on an input parameter. However, several of the loops were conditionally taken or not taken depending on some input parameters.

The remaining columns in Table 3 summarize the results of the infeasible path analysis. **Time** gives the analysis time for the loop bound and infeasible path analyses in seconds, and **Cov** gives the percentage of NIC basic blocks abstractly executed by the AE. As expected, the time for the AE grows with the size of the program. The number of flow facts for infeasible nodes **IN**, excluding node pairs **EP**, longer infeasible paths **IP**, and maximal node count **MC** derived by SWEET are also given. All tasks, except task 10, were analyzed with merge points set at loop- and function exits. Task 10 contained code parts with many conditions, causing the number of abstract states generated by the AE to grow too fast for this merging strategy. By using a higher degree of merging, with merge points additionally set after if-statements, also task 10 could be analyzed. However, this meant that only **IN** and **MC** flow facts could be derived for this task. Therefore, the **EP** and **IP** columns are marked by "-" for this task.

An initial analysis of task 13 indicated the same problems as for task 10. The number of inputs affecting the flow was also large, making the size of each abstract state large. Moreover, the analysis of task 13 posed a problem due to the large amount of globals that needed manual input value annotations. The globals were stored in many different C files and many were of state variable type. The task was analyzed at the end of the project, leaving too little time to do all the needed annotations. As a result, no infeasible path analysis was made for this task, which is indicated by "-" in the corresponding columns.

The automatically generated infeasible path flow facts were manually translated to aiT flow annotations. The number of such annotations are given in **DA**. Some of the generated flow facts were redundant and were therefore not translated. The context-sensivity inherent in the SWEET flow facts also made some translations fail. We made two runs with aiT, one using manually given flow annotations from the previous case study [37] and one using the translated SWEET flow facts. For all the successfully analyzed tasks 1–12, aiT produced exactly the same WCET estimates. We also visually compared the two aiT generated worst-case path graphs to make sure that were equal. We therefore conclude that for these tasks SWEET was able to produce infeasible path information which tightened the generated WCET estimate equally well as the previously manually given flow annotations.

## 10 Conclusions and Future Work

We have presented a case study where an automatic flow analysis method was used for WCET analysis of industrial production code. The analysis method was for the most part able to automatically derive flow information of at least the same quality as the earlier, manually provided information [37], and the resulting WCET estimates were identical. This provides evidence that automatic flow analysis methods can reduce the need for manual annotations, thus making WCET analysis tools easier to use.

For the studied code, loop iteration bounds turned out to mostly not be dependent on input data. However, infeasible paths were. Thus, to obtain tight WCET estimates, flow analysis methods should be input-sensitive. Further, the code had many inputs, provided through complex data structures. To provide correct value constraints for all these input variables was not always an easy task. Academic benchmark code for WCET analysis mostly has a few rather simple input variables, thus not exposing this problem.

The studied code has state variables which are persistent between task invocations. We believe this is common for task-oriented code. For the studied code these variables often affect the program flow, thus better WCET estimates can potentially be obtained by bounding their values. This is not always easy, however, since their possible values for new task invocations recursively depend on their previous values. We designed and implemented the *State Variable Value Analysis* (SVVA) to automatically bound the values of such variables. The current version of SVVA only concerns persistent variables pertaining to a single, nonpreemptive task. We plan to study how to extend SVVA to systems of communicating, possibly preempted tasks, also taking into account possible restric-

tions on the invocation order among tasks.

However, there will always be variables whose values will not be possible to bound by an automatic code analysis, like data originating from the outside world. Therefore, it is also important with better means to provide bounds on data manually when needed.

For two tasks, we failed to derive precise infeasible path constraints automatically. The reason for this was primarily the memory consumption of the analysis: both these tasks have many conditionals and a very large number of possible paths. Our current merge strategies during AE does not seem to reduce the number of abstract states optimally, which may result in a large number of abstract states simultaneously held in memory. Furthermore, the sometimes large data areas, and the inability of our slicing to reduce these areas much, led to single abstract states being large. We believe these are the two main reasons why the analysis failed for these tasks.

There are several ways to reduce the memory needs. Our current slicing does not distinguish between elements in NIC data areas (corresponding to structs or arrays in C). Thus, it is possible that the program flow in the analyzed tasks actually only depends on small parts of their respective input data areas. A more precise slicing, differing between different parts of the areas, could thus potentially reduce the size of the needed abstract states: this would also reduce the need for manual value annotations in cases where automatic methods for deriving these won't work. Sharing between abstract states could be improved: a promising idea builds on *version arrays* for efficient update of arrays in side-effect-free languages [1]. If elements in data areas turn out to mostly hold the same value, then *compressed state representations* can be beneficial.

We currently consider *strategies for lower the number of simultaneous abstract states*, e.g., by introducing ordered merge, meaning that merged states "further from" termination are released first.

Finally, we have recently developed loop- and infeasible path analyses based on a combination of *program slicing, value analysis and invariant analysis* [17]. These methods trade precision for better worst-case behaviour, and they should also in general have less need for memory.

## References


[1] A. Aasa, S. Holmström, and C. Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28(3):489–503, Sept. 1988.

[2] AbsInt. aiT tool homepage, 2008. `www.absint.com/ait`.

[3] H. Aljifri, A. Pons, and M. Tapia. Tighten the computation of worst-case execution-time by detecting feasible paths. In *Proc. 19$^{th}$ IEEE International Performance, Computing, and Communications Conference (IPCCC2000)*. IEEE, Feb. 2000.

[4] P. Altenbernd. On the false path problem in hard real-time programs. In *Proc. 8$^{th}$ Euromicro Workshop of Real-Time Systems*, pages 102–107, June 1996.

[5] D. Barkah. Evaluating program flow analysis for WCET calculations at Volvo CE. Technical report, Mälardalen University, Västerås, Sweden, Aug. 2007.

[6] S. Bygde. Analysis of arithmetical congruences on low-level code. In *Nordic Workshop on Programming Theory '07*, pages 47–48, Oslo, Norway, Oct. 2007.

[7] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. Applying static WCET analysis to automotive communication software. In *Proc. 17$^{th}$ Euromicro Conference of Real-Time Systems, (ECRTS'05)*, July 2005.

[8] M. Carlsson, J. Engblom, A. Ermedahl, J. Lindblad, and B. Lisper. Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system. In *Proc. 2$^{nd}$ International Workshop on Real-Time Tools (RT-TOOLS'2002)*, 2002.

[9] T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra. Exploiting branch constraints without exhaustive path enumeration. In *Proc. 5$^{th}$ International Workshop on Worst-Case Execution Time Analysis, (WCET'2005)*, pages 40–43, July 2005.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4$^{th}$ ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, Jan. 1977.

[11] Copy-on-write explanation. Wikipedia, the free encyclopedia. `en.wikipedia.org/wiki/Copy-on-write`.

[12] C. Cullmann and F. Martin. Data-flow based detection of loop bounds. In *Proc. 7$^{th}$ International Workshop on Worst-Case Execution Time Analysis, (WCET'2007)*, pages 57–62, Pisa, Italy, July 2007.

[13] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala, Sweden, Apr. 2002. ISBN 91-554-5228-0.

[14] O. Eriksson. Evaluation of static time analysis for CC systems. Master's thesis, Mälardalen University, Sweden, Aug. 2005. 63 pages, `www.mrtc.mdh.se/publications/0978.pdf`.

[15] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden, June 2003.

[16] A. Ermedahl, J. Gustafsson, and B. Lisper. Experiences from industrial WCET analysis case studies. In *Proc. 5$^{th}$ International Workshop on Worst-Case Execution Time Analysis, (WCET'2005)*, pages 19–22, July 2005.

[17] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation,

and invariant analysis. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis, (WCET'2007)*, Pisa, Italy, July 2007.

[18] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. 1st International Workshop on Embedded Systems, (EMSOFT2000), LNCS 2211*, Oct 2001.

[19] C. Ferdinand, R. Heckmann, and H. Theiling. Convenient user annotations for a WCET tool. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, 2003.

[20] C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. New developments in WCET analysis. In *Program Analysis and Compilation, Theory and Practice*, volume 4444 of *Lecture Notes in Comput. Sci.*, pages 12–52. Springer-Verlag, 2007.

[21] J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Dept. of Information Technology, Uppsala University, Sweden, May 2000.

[22] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system C programs. In *Proc. 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005)*, Feb. 2005.

[23] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27th IEEE Real-Time Systems Symposium (RTSS'06)*, Dec. 2006.

[24] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, 18(2-3):129–156, May 2000.

[25] C. Healy and D. Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, jun 1999.

[26] N. Holsti, T. Långbacka, and S. Saarinen. Using a worst-case execution-time tool for real-time verification of the DEBIE software. In *Proc. DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457)*, Sept. 2000.

[27] N. Holsti, T. Långbacka, and S. Saarinen. Worst-case execution-time analysis for digital signal processors. In *Proc. EUSIPCO 2000 Conference (X European Signal Processing Conference)*, 2000.

[28] Infineon. Infineon Systems homepage, 2006. `www.infineon.com`.

[29] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, June 2002.

[30] Mälardalen University. WCET project homepage, 2007. `www.mrtc.mdh.se/projects/wcet`.

[31] P. Montag, S. Goerzig, and P. Levi. Challenges of timing verification tools in the automotive domain. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, Paphos, Cyprus, Nov. 2006.

[32] RapiTime WCET tool homepage, 2006. `www.rapitasystems.com`.

[33] M. Rodriguez, N. Silva, J. Esteves, L. Henriques, D. Costa, N. Holsti, and K. Hjortnaes. Challenges in calculating the WCET of a complex on-board satellite application. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, 2003.

[34] Rubus homepage, 2006. `www.arcticus-systems.com/productsrubusos.php`.

[35] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper. Faster WCET flow analysis by program slicing. In *Proc. ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'06)*, pages 103–112, June 2006.

[36] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static timing analysis of real-time operating system code. In *Proc. 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, Oct. 2004.

[37] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz. Static WCET analysis of real-time task-oriented code in vehicle control systems. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, Paphos, Cyprus, Nov. 2006.

[38] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[39] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *Proc. of the IEEE Int. Conf. on Dependable Systems and Networks (DSN-2003)*, June 2003.

[40] Tidorum. Bound-T tool homepage, 2006. `www.tidorum.fi/bound-t`.

[41] Volvo CE (construction equipment) homepage, 2008. `www.volvo.com/constructionequipment`.

[42] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[43] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem — overview of methods and survey of tools. *Accepted for publication in ACM Transactions on Programming Languages and Systems*, 2008.

[44] The Whole-Program Optimization project homepage, 2001. `www.astec.uu.se/etapp3/`.