Towards a Flow Analysis for Embedded System C Programs

Jan Gustafsson, Andreas Ermedahl, and Björn Lisper
Dept. of Computer Science and Electronics, Mälardalen University, Västerås, Sweden
{jan.gustafsson,andreas.ermedahl,bjorn.lisper}@mdh.se

Abstract

Reliable program Worst-Case Execution Time (WCET) estimates are a key component when designing and verifying real-time systems. One way to derive such estimates is by static WCET analysis methods, relying on mathematical models of the software and hardware involved. This paper describes an approach to static flow analysis for deriving information on the possible execution paths of C programs. This includes upper bounds for loops, execution dependencies between different code parts and safe determination of possible pointer values. The method builds upon abstract interpretation, a classical program analysis technique, which is adopted to calculate flow information and to handle the specific properties of the C programming language.

1. Introduction

A *Worst-Case Execution Time* (WCET) analysis finds an upper bound to the worst possible execution time of a computer program. Reliable WCET estimates are crucial when designing and verifying embedded and real-time systems, especially when such systems are used to control safety-critical systems like vehicles, military equipment and industrial power plants. These estimates are needed in hard real-time systems development for schedulability analysis, to determine if performance goals are met for periodic tasks, and to check that interrupts have sufficiently short reaction times [19].

The traditional way to determine the timing of a program is by measurements. A wide variety of measurement tools are employed in industry, including emulators, logic analyzers, oscilloscopes, and software profiling tools [34]. This is labor-intensive and error-prone work, and even worse, it cannot guarantee that the true WCET has been found since, in general, it is impossible to perform exhaustive testing.

Static WCET analysis determines the WCET of a program, relying on mathematical models of the software and hardware involved. The analysis avoids the need to run the program by considering the effects of all possible inputs, including possible system states, together with the program's interaction with the hardware. Given that the models are correct, the analysis will derive a timing estimate that is *safe*, i.e., greater than or equal to the actual WCET.

To be able to statically estimate the WCET for a program, information on both the program's *hardware interaction* (to get the timing of different instructions) and on possible *program execution flows* (to bound the number of times the instructions can be executed) needs to be derived.

C is currently the dominant language for embedded system software development [33], and has many features suitable for embedded system development, such as powerful pointer and bit operations and access to the underlying data representation. Unfortunately, these features makes it hard to apply flow analysis techniques to C.

This paper presents ongoing work in the development of automatic flow analysis methods to analyse real-world embedded C programs. We use abstract interpretation [10], a formal program analysis technique to derive properties of the analysed program. During our work, we have developed the abstract interpretation along three main lines:

1. We use abstract interpretation to calculate flow information, such as upper bounds to loop iteration counts.
2. We have created an abstract analysis domain to handle the data representation in C. This allows us to handle C features like structs, arrays, pointers and type casts.
3. We do not perform our analysis directly on the C source code. Instead, it is applied on an intermediate code format, making our flow analysis more generic and less dependent on C source characteristics.

The rest of this paper is organized as follows: In Section 2, we given an introduction to static WCET analysis and present some related work. In Section 3 we present classical abstract interpretation. In Section 4 we extend the abstract interpretation to calculate flow information, and in Section 5 we extend the abstract domain to support data types and structures in C. Section 6 presents our WCET tool prototype, SWEET (SWEdish Execution time Tool). Finally, in Section 7 we draw some conclusions and present our ideas for future work.

2. Static WCET Analysis Overview

Any WCET analysis must deal with the fact that most computer programs do not have a fixed execution time. *Variations* in the execution time occur due to different input data, the characteristics of the software, as well as of the hardware upon which the program is run.

Consequently, static WCET analysis is usually divided into three phases: a (fairly) machine-independent *flow analysis* of the code, where information about the possible program execution paths is derived, a *low-level analysis* where the execution time for atomic parts of the code is decided from a performance model for the target architecture, and a final *calculation* phase where flow and timing information previously derived are combined to yield a WCET estimate.

The purpose of the flow analysis phase is to extract constraints on the dynamic behavior of the program. This includes information on which functions get called, loop bounds, dependencies between conditionals, etc. Since the flow analysis does not know the execution path which corresponds to the longest execution time, the information must be a safe (over)approximation including *all* possible program executions. The information can be obtained by *manual annotations* (integrated in the programming language [26] or provided separately [15, 17]), or by *automatic flow analysis* methods [21, 23, 25, 27].

The purpose of low-level analysis is to determine the timing behaviour of instructions, given the architectural features of the target hardware. For modern processors it is especially important to study the effects of various performance enhancing features, like caches, branch predictors and pipelines [3, 13, 24, 27].

The purpose of the calculation phase is to calculate a WCET estimate, combining the flow and timing information derived in the previous phases. A frequently used calculation method is IPET (Implicit Path Enumeration Technique), using arithmetical constraints to model the program flow and low-level execution times [15, 18, 25].

The presented work is carried out within a project with the research goal to develop WCET analysis methods for real-time embedded systems. Our current research focus is on automatic flow analysis methods. The need for such methods has been further motivated by recent case-studies on static WCET analysis with embedded system vendors [7, 8, 32]. We have found it especially important is to focus on automatic methods for loop bound analysis, since these annotations have been found particularly troublesome to give.

3. Classical Abstract Interpretation

Abstract interpretation was presented in the 1970s by Cousot and Cousot [10]. It is a formal framework for program analysis which guarantees *correctness*, in the sense that a predicted program property is surely true. It uses a generic solution method called *fixed-point iteration* which will, under some conditions, yield the answer in finite time. The original framework by Cousot and Cousot is defined for flowcharts (basically control flow graphs) describing imperative, possibly unstructured programs. The framework can be extended to (possibly recursive) functions.

Abstract interpretation has three important properties:

1. It yields an *approximate* and *safe* description of the program behavior.
2. It is *automatic*, i.e., the program does not have to be annotated.
3. It works for *all* programs in the selected language.

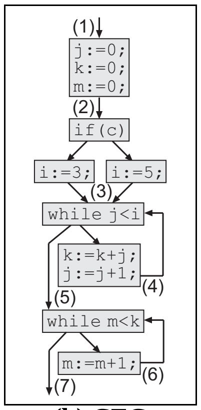
Abstract interpretation can be used to statically determine run-time information of many varieties. One use has been to calculate possible values for variables at different program points. This information can for example be used in a ”static debugger” [6] which identifies (the risk of) array indices lying outside array ranges, and other possible errors in the analysed program. This idea has been further developed and commercialised under the name PolySpace [29]. Recent examples of the use of abstract interpretation include special-purpose program analyzers for verification of large embedded real-time software [4, 5, 35]. Another use has been to automatically find complexity measures for programs [30]. Abstract interpretation is also used in in the commercial WCET tool aiT [1] as a main technique in the various program analyses performed in the tool.

Abstract interpretation uses *abstract states*, which represent sets of program states. Each flowchart node is given an *abstract transition function*, which maps abstract states to abstract states. The set of abstract states $\mathcal{S}$ must form a *complete lattice* $\langle \mathcal{S}, \sqcup, \sqcap, \sqsubseteq, \top, \bot \rangle$. $\sqsubseteq$ is an ordering relation corresponding to the subset relation on the corresponding sets of states. $\sqcup, \sqcap$ correspond to union and intersection, respectively, but may overapproximate the corresponding operation. $\sqcup$ is often called “merge”, and typically appears in transfer functions, for nodes with several incoming edges, to merge their abstract states. Finally, $\bot$ is the *bottom element* (w.r.t. $\sqsubseteq$), representing $\emptyset$, and $\top$ is the *top element*, representing the universal set. If $S \sqsubseteq S'$, then S yields more precise information about the possible states than S'. The top element yields no information at all.

Program analysis by abstract interpretation assigns an abstract state S^p to each program point p. The abstract transition functions define a system of equations $\vec{S} = \vec{F}(\vec{S})$ for the vector S of the abstract states in the different program points. If certain axioms are met, then a solution can always be found by fixed-point iteration: $\vec{S}_i = \vec{F}(\vec{S}_{i-1})$. The iteration starts with the least possible assignment $\vec{S}_0 = \vec{\bot}$, assigning the bottom element to each program point, and

```
1) j := 0;
   k := 0;
   m := 0;
2) if (c)
      i := 3;
   else
      i := 5;
3) while j < i do
      k := k + j;
      j := j + 1;
4) end do
5) while m < k do
      m := m + 1;
6) end do
7) ...
```

(a) Example program



(b) CFG

$$\begin{aligned}
S^1 &= [i \mapsto \bot, j \mapsto \bot, k \mapsto \bot, c \mapsto [-\infty..+\infty], m \mapsto \bot]\\
S^2 &= [i \mapsto \bot, j \mapsto [0..0], k \mapsto [0..0], c \mapsto [-\infty..+\infty], m \mapsto [0..0]]\\
S^3 &= [i \mapsto [3..5], j \mapsto [0..0], k \mapsto [0..0], c \mapsto [-\infty..+\infty], m \mapsto [0..0]]\\
S^4 &= S^5 = S^6 = S^7 = [i \mapsto \bot, j \mapsto \bot, k \mapsto \bot, c \mapsto \bot, m \mapsto \bot]
\end{aligned}$$

(c) Abstract states at the beginning of the first loop

$$\begin{aligned}
S^1 &= [i \mapsto \bot, j \mapsto \bot, k \mapsto \bot, c \mapsto [-\infty..+\infty], m \mapsto \bot]\\
S^2 &= [i \mapsto \bot, j \mapsto [0..0], k \mapsto [0..0], c \mapsto [-\infty..+\infty], m \mapsto [0..0]]\\
S^3 &= [i \mapsto [3..5], j \mapsto [0..+\infty], k \mapsto [0..+\infty], c \mapsto [-\infty..+\infty], m \mapsto [0..0]]\\
S^4 &= [i \mapsto [3..5], j \mapsto [1..5], k \mapsto [0..+\infty], c \mapsto [-\infty..+\infty], m \mapsto [0..+\infty]]\\
S^5 &= [i \mapsto [3..5], j \mapsto [3..+\infty], k \mapsto [0..+\infty], c \mapsto [-\infty..+\infty], m \mapsto [0..+\infty]]\\
S^6 &= [i \mapsto [3..5], j \mapsto [3..+\infty], k \mapsto [0..+\infty], c \mapsto [-\infty..+\infty], m \mapsto [0..+\infty]]\\
S^7 &= [i \mapsto [3..5], j \mapsto [3..+\infty], k \mapsto [0..+\infty], c \mapsto [-\infty..+\infty], m \mapsto [0..+\infty]]
\end{aligned}$$

(d) Final abstract states after using widening

Figure 1. Example of integer interval analysis

will then produce the *least* assignment solving the equations. This gives the best possible precision in the program analysis.

For some abstract interpretations the fixed-point iteration will not always terminate. Termination can be guaranteed through a binary *widening operator* $\triangledown$ on abstract states, obeying some axioms [10]. For some program point p, the recursive equation $S_i^p = F^p(\vec{S}_{i-1}^{in})$ is changed to $S_i^p = S_{i-1}^p \triangledown F^p(\vec{S}_{i-1}^{in})$. If at least one equation along each cycle in the flowchart is changed in this way, then termination is ensured. The solution obtained when iterating the new equations will be correct, but it might not be the least one.

The approximative solution can be improved using a *narrowing operator* $\triangle$. It can be shown that the downward abstract iterative sequence using narrowing $S_i^p = S_{i-1}^p \triangle F^p(\vec{S}_{i-1}^{in})$ will yield an upper approximation of the fixpoint that is better than the result found by widening, under some circumstances [11].

3.1. Abstract Interpretation using Intervals

In [9], Cousot proposed the following abstract domain using the lattice of integer intervals

$$\{\bot\} \cup \{[l..u] \mid l \in \mathcal{Z} \cup \{-\infty\} \wedge u \in \mathcal{Z} \cup \{+\infty\} \wedge l \leq u\}$$

as an approximation of sets of integers, and ordered by set inclusion. This approximation is formalised by a pair of functions (the *Galois connection*) defined by

$$\begin{aligned}
\gamma(\bot) = \emptyset, &\quad \gamma([l..u]) = \{x \in \mathcal{Z} \mid l \leq x \leq u\}\\
\alpha(\emptyset) = \bot, &\quad \alpha(X) = [min(X)..max(X)]
\end{aligned}$$

The *abstraction* function α creates an interval $[l..u]$ which is a safe aproximation of a set of integers. The *concretization* function γ creates a set of concrete values (integers) from an abstract value.

Example. We will present an example using this abstraction. In Section 4 we develop a different way than presented here to set up and solve a system of equations for abstract states. The example motivates this development. Consider the very small program fragment P shown in Figure 1(a), with its control flow graph (CFG) in Figure 1(b). P is written in a generic imperative programming language, using only integer variables.

We are interested in the loop bounds of the two while-loops. The two variables `j` and `m` behave as loop counters; if the analysis can find possible values of these, the result could be used as loop bounds. Also, we notice that the number of iterations in the second loop is dependent on the first. We will show that the "classical" abstract interpretation is unable to calculate the number of iterations in the second loop.

We will visualise an abstract state S^p with the variable mapping at the program point p. For increased clarity we will only annotate a few program points (marked 1, …, 7 in Figure 1) in this way during analysis. Also we will not, for space reasons, show the details of the calculations.

We have the initial state $\lambda x.\bot$ at all program points. After a few iterations in the fixed-point calculation we are at the beginning of the first loop, with the abstract states in Figure 1(c). We assume that `c` is an unknown input parameter, therefore it is set to the top ($\top$) value $[-\infty..+\infty]$ prior to this code. Note that `i` has received the value [3..5] due to the merge of the if-edges and because `c` is unknown.

After a number of iterations of the fix-point calculation, `j` will stabilize to the final value 5 at the join point at the beginning of the first while loop. The other variable updated in the loop, `k`, (i.e., an induction variable), will not find a stable value but grow infinitely. Thus, the calculation will not terminate; one reason for this is that the domain contains infinitely ascending chains.

As mentioned above, one way to enforce termination is to use a widening operator. In [9], Cousot introduced the following widening operator for intervals:

$$\begin{aligned}
&\bot \bigtriangledown X = X \bigtriangledown \bot = X \\
&[l_0..u_0] \bigtriangledown [l_1..u_1] = \\
&\quad [(\text{if } l_1 < l_0 \text{ then } -\infty \text{ else } l_0).. \\
&\quad (\text{if } u_1 > u_0 \text{ then } +\infty \text{ else } u_0)]
\end{aligned}$$

If we use this widening operator at the join program points for the loops (i.e., program points 3 and 5 in Figure 1), the fix-point calculation will terminate, with the result in Figure 1(d). As a result of the widening, all unstable bounds are extended to infinity. We see that `j` has the value $[1..5]$ in S^4 inside the first loop, i.e., the upper loop bound of the first loop is 5. However, we have no information about the second loop bound, since `m` is unbound in S^6.

We can try to improve this result by applying the narrowing operator below, also defined in [9]:

$$\begin{aligned}
&\bot \bigtriangleup X = X \bigtriangleup \bot = \bot \\
&[l_0..u_0] \bigtriangleup [l_1..u_1] = \\
&\quad [(\text{if } l_0 = -\infty \text{ then } l_1 \text{ else } l_0).. \\
&\quad (\text{if } u_0 = +\infty \text{ then } u_1 \text{ else } u_0)]
\end{aligned}$$

Applying this narrowing in a downward abstract iteration will reduce `j` to $[3..5]$ in S^5, S^6 and S^7, but we get no better results for `k` and `m`, i.e., we do not get any loop bound on the second loop.

The conclusion from this example is that classical abstract interpretation can not calculate all loop bounds. We have developed abstract interpretation further to calculate additional loop bounds, as described in next section.

4. Extending Abstract Interpretation for Flow Fact Extraction

Classical abstract interpretation merges program states whenever the control flow joins, analyses all iterations for a loop "collectively", and enforces fast termination using widening. Narrowing is used to reduce the overestimation of values. As we saw, in the example in Figure 1, classical abstract interpretation cannot calculate all loop bounds, but has to developed further to be able to calculate more bounds.

We have suggested a type of abstract interpretation (a kind of "abstract execution" described in more detail in, e.g., [16, 21]) where the loops are "rolled out" dynamically and each iteration is analysed individually.

In the basic version of the method, no merge is done at join program points. Instead, all feasible paths in the program are analyzed independently.

Example revisited. With that type of analysis, we analyze the loops in Figure 1 as shown in Table 1 and 2. As mentioned, the paths are analyzed independently. Since the flow forks after program point 2, both loops will be analyzed twice. The tables show the analysis of the first and the second loop, respectively. The first part of the tables shows the analysis for the path where $i = [3..3]$, and the second part where $i = [5..5]$. We see that each iteration is analysed individually.

In this very simple example, the ranges are just single values. Normally, the abstract values are ranges representing many values.

Table 1. Analysis of the first loop.

S^i	iter.	i	j	k	c	m
3	-	$[3..3]$	$[0..0]$	$[0..0]$	$[-\infty..+\infty]$	$[0..0]$
4	1	$[3..3]$	$[1..1]$	$[0..0]$	$[-\infty..+\infty]$	$[0..0]$
4	2	$[3..3]$	$[2..2]$	$[1..1]$	$[-\infty..+\infty]$	$[0..0]$
4	3	$[3..3]$	$[3..3]$	$[3..3]$	$[-\infty..+\infty]$	$[0..0]$

S^i	iter.	i	j	k	c	m
3	-	$[5..5]$	$[0..0]$	$[0..0]$	$[-\infty..+\infty]$	$[0..0]$
4	1	$[5..5]$	$[1..1]$	$[0..0]$	$[-\infty..+\infty]$	$[0..0]$
4	2	$[5..5]$	$[2..2]$	$[1..1]$	$[-\infty..+\infty]$	$[0..0]$
4	3	$[5..5]$	$[3..3]$	$[3..3]$	$[-\infty..+\infty]$	$[0..0]$
4	4	$[5..5]$	$[4..4]$	$[6..6]$	$[-\infty..+\infty]$	$[0..0]$
4	5	$[5..5]$	$[5..5]$	$[10..10]$	$[-\infty..+\infty]$	$[0..0]$

Table 2. Analysis of the second loop.

S^i	iter.	i	j	k	c	m
5	-	$[3..3]$	$[3..3]$	$[3..3]$	$[-\infty..+\infty]$	$[0..0]$
6	1	$[3..3]$	$[3..3]$	$[3..3]$	$[-\infty..+\infty]$	$[1..1]$
6	2	$[3..3]$	$[3..3]$	$[3..3]$	$[-\infty..+\infty]$	$[2..2]$
6	3	$[3..3]$	$[3..3]$	$[3..3]$	$[-\infty..+\infty]$	$[3..3]$

S^i	iter.	i	j	k	c	m
5	-	$[5..5]$	$[0..0]$	$[10..10]$	$[-\infty..+\infty]$	$[0..0]$
6	1	$[5..5]$	$[5..5]$	$[10..10]$	$[-\infty..+\infty]$	$[1..1]$
6	2	$[5..5]$	$[5..5]$	$[10..10]$	$[-\infty..+\infty]$	$[2..2]$
6	3	$[5..5]$	$[5..5]$	$[10..10]$	$[-\infty..+\infty]$	$[3..3]$
6	4	$[5..5]$	$[5..5]$	$[10..10]$	$[-\infty..+\infty]$	$[4..4]$
6	5	$[5..5]$	$[5..5]$	$[10..10]$	$[-\infty..+\infty]$	$[5..5]$
6	6	$[5..5]$	$[5..5]$	$[10..10]$	$[-\infty..+\infty]$	$[6..6]$
6	7	$[5..5]$	$[5..5]$	$[10..10]$	$[-\infty..+\infty]$	$[7..7]$
6	8	$[5..5]$	$[5..5]$	$[10..10]$	$[-\infty..+\infty]$	$[8..8]$
6	9	$[5..5]$	$[5..5]$	$[10..10]$	$[-\infty..+\infty]$	$[9..9]$
6	10	$[5..5]$	$[5..5]$	$[10..10]$	$[-\infty..+\infty]$	$[10..10]$

Since the respective loop variables are initialized to zero and incremented by one, the respective interval for the final iteration bounds the maximal iteration count for the loop in question. Thus, we conclude that the possible number of iterations, for the first loop in Figure 1, is in the range $[3..5]$, and for the second loop in the range $[3..10]$. In this way, we can find better loop bounds than with classical abstract interpretation:

- We get both lower and upper loop bounds for the second loop.
- We get a tighter lower loop bound for the first loop.

Another advantage is that we can get results for individual loop iterations. For example, when having nested loops, we can get individual loop bounds for the inner loops for each iteration of the outer loops.

Our general approach to calculate loop limits is to instrument the loops with *execution counter variables* that are set to 0 prior to the loop, incremented by one in the loop, and then use our type of abstract interpretation to find the final values of these variables. It is worth mentioning, that these execution counter variables will behave as induction variables, i.e., like the variable `k` in Figure 1, and as we have pointed out, classical abstract interpretation can not calculate the bounds for these. There are, however, other methods to do calculations for some types of induction variables, like [20], but our type of of abstract interpretation is a more general solution.

The loop bounds are, as well as information on infeasible paths, recursion depth etc., information on the possible flows in the program. We call this information *flow facts*, and use a specially developed *flow fact language* to express this information. The *scope graph* is a program representation where these flow facts can be stored [14].

4.1. Merging

One consequence of this type of abstract interpretation is that each iteration up to the loop bound has to be analyzed. This means of course that quite heavy calculations have to be made. There is a risk of state explosion since there can be a lot of alternatives in each iteration.

To reduce this problem we introduce *merging* of abstract states at certain join program points. We have developed four types of merge strategies, based on the selection of different types of merge points:

- Type 1: merge at *function end*. The merge points are the targets of the return statements in functions.
- Type 2: merge at *loop termination*. The merge points are the termination points after loops.
- Type 3: merge at *loop body* end. Merging is performed at the header after each loop iteration.
- Type 4: merge at *if*. The merge points are the join points after selection statements.

Merging may of course introduce some overestimations. For example, if a merge is done at the join point after an if statement, the results of both branches will be merged into one, and we may include some values in the result which are not possible in real executions. However, it can be shown that the result is safe in the sense discussed in Section 3.

Consider for example the example in Figure 1 again. Doing a merge after if statements (type 4) would have lead to a *single* analysis of the first loop with $i = [3..5]$. We would have included the infeasible value $i = 4$ in the calculations, but the result would still be safe, and we would have less calculations in the analysis.

As an another example, consider an analysis of the following code fragment: `if(x<3)` S_1 `else` S_2`;` `if(x>17)` S_3 `else` S_4. Without merging, we obtain four different abstract states after the last statement. However, the abstract state corresponding to the sequence S_1-S_3 becomes infeasible since `x` cannot both be smaller than 3 and larger than 17 (assuming that `x` is not changed in S_1). We can therefore conclude that S_1-S_3 can not be executed and therefore is an infeasible path. If we would have merged the abstract states after each if statement (type 4 above) we would have lost this information.

So, there is a trade-off between analysis speed and precision of the result. The different types of merging allows us to make experiments to choose the optimal merging strategy for a certain piece of code.

5. Extending the Abstract Domain for C

Most embedded systems are programmed in C and/or assembly language. More sophisticated languages, such as C++, Ada or Java, have found some use, but the need for speed, portability, small code size, and efficient access to the hardware has kept C the dominant language in such systems [33]. Furthermore, embedded software is often specialized for a particular hardware platform and a special purpose, and the code can therefore differ quite significantly from ordinary desktop code [12]. For example, features like unstructured code, deeply nested loops, function pointers, logical and bitwise operations are common in embedded real-time code, while dynamical features like dynamical allocation of memory and recursion are used to a lesser extent. A flow analysis suitable for embedded system C code must be able to handle all common features in an efficient and safe manner.

5.1. Analysing C programs

An analysis of C programs requires an adaption of the classical abstract interpretation outlined in Section 3. In C, memory representation of data is very explicit, i.e., all values can be seen as a sequences of bits, and the interpretation depends on the operations applied on the values. The data can be stored in memory and referenced using pointers. For example, `int* p = &i` assigns the address of the `i` variable to the pointer `p`. The address can, similar to other values, be seen as a sequence of bits on which operations can be applied. This explicit representation of data is suitable for embedded programmers who often need to manipulate and access their data on a very fine-grained level. For example, memory mapped I/O requires access to dedicated memory addresses.

Furthermore, due to memory constraints the embedded system C programmer often tries to use minimal sized data types, e.g., using chars or shorts instead of ints whenever possible [12]. These data types are limited in their value ranges. For example, an unsigned char can only hold a value between 0 and 255. This means that the abstract value domains must be adapted to be suitable for the data types used in C, i.e., instead of $[-\infty..+\infty]$ a variable of unsigned char type should have a possible value domain of $[0..255]$. Furthermore, C has no protection for arithmetical overflow of values, making the code `unsigned char c = 255; c = c + 1;` result in the value 0, a fact that the analysis must handle correctly.

In C a chunk of data, like an array, is actually a continuous piece of memory where you can access individual elements using a start address and an offset. For example, an array `char A[5]` can be referenced using normal array indexing: `char c = A[2];` or using a pointer: `char* p = A; c = *(p + 2);`. Furthermore, you can read smaller parts of a datum or you can merge several smaller parts into a larger piece of data. For example, `int i = *((int *)(A+1));` merges the four last chars in the `A` array and stores the resulting value as a 32-bit integer value in the variable `i`. C is strongly typed, but you can (implicitly or explicitly) create data of a specific type from a data of another type.

C has powerful operations for bit manipulation of data, like left (<<) and right (>>) bitshift, binary or (`|`) and binary and (&). For the embedded system programmer, such operations can be useful for setting and manipulating individual bits in ports or dedicated registers. However, these operations makes it hard to represent possible data values only as intervals, since we also need to know how the data is represented at bit level. For example, when performing a binary `k = j & 25;` operation, an interval representation of `k` will be dependent of the possible bitpatterns of the interval representation of `j`.

The bit representation of data becomes even more apparent when we consider how chars, shorts and integers are represented. Basically, for each such data you can interpret the bitstring as either signed or unsigned. This means that it might be the case that `(i < j)` is true if `i` and `j` both are signed, but not true if they are both unsigned. Furthermore, when comparing values of different types, C makes implicit type conversions. For example, assuming `i` is a signed char with value `-1` and `j` is an unsigned int with value `5`, a comparison `(i < j)` would be evaluated to false. This is because `i` is interpreted as a unsigned int, which becomes a very large number, before being compared. Since type conversions (both implicit and explicit) are frequently used in C the abstract interpretation must be able to handle this.

Pointers are frequently used by embedded system C programmers. They give the programmer large freedom, but are also a common source for programming errors. Pointers also cause problems for the classical interval analysis. A straightforward interval representation of a pointer would be an interval covering all possible addresses which the pointer could hold. However, this is not possible on the C code level since it is not defined at what particular memory addresses data and instructions should be stored, i.e., this is left for the compiler and linker to decide. Therefore, a safe analysis assumption would be that every pointer could hold any address of every data in the program. This would however be overly pessimistic in most cases and result in non-usable analysis results.

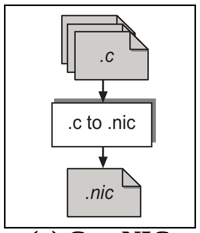
Unfortunately, a flow analysis suitable for C must handle pointers and pointer operations, since these are frequently used to decide the outcome of conditionals. For example, Sandberg [31] reports that more than 50% of the conditionals of the analyzed embedded system codes were directly dependent on pointers. Pointers can also be used to indirectly manipulate data which control conditionals. For example, in `int* p = &i; *p = 5; if(i < j)` the `p` pointer is used to assign a value to the `i` variable, and is indirectly deciding the outcome of the branch condition.

5.2. The NIC format

One intricate question is to which program code level that derived flow information should be related. Flow information can be provided in relation to the source code, the intermediate code in a compiler, or to the object code. For WCET-tool users, flow information or value constraints on variables are often easier to provide at the source-code level. However, such flow information must somehow be *mapped* down to the object code to be used in the WCET calculation. In the presence of optimizing compilers, this mapping problem is non-trivial since transformations like loop unrolling, function inlining and code duplication can be performed [28]. On the other hand, automatic flow analysis is harder to perform at the object code level since variables and other entities of interest are harder to identify in the object code.

We perform our flow analysis on NIC (New Intermediate Code), an intermediate code format designed for embedded system code analysis and compilation. NIC and its supporting environment has been developed by a research project at Uppsala University targeting whole program analysis [37]. NIC is generated from C and is able to handle (almost) complete ANSI-C. By performing the flow analysis on an intermediate code format we will be able to evaluate the benefits of integrating static WCET analysis into a compiler framework.

NIC explicitly expresses most of the things that are implicit in C. For example, when comparing a signed char and an unsigned int a lot of implicit type conversions are made in C, but in NIC all these steps are explicitly represented by NIC instructions. NIC can represent the C source code



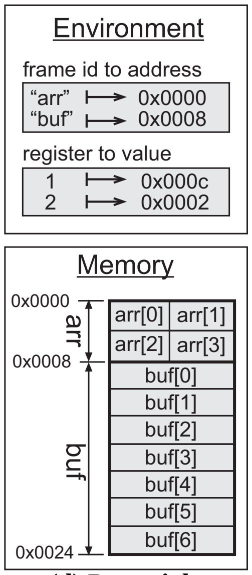
(a) C to NIC conversion

```
int foo(void)
{
 short arr[4];
 int i = 2;
 int buf[7];
 int *p;
 buf[5] = i;
 p = buf;
 p++;
 ....
}
```

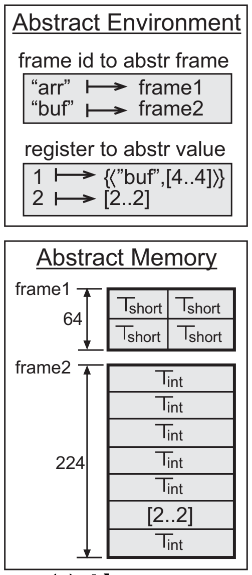
(b) Example C code fragment

```
...
(function ()
(register_manager
 ((register 1 32) (annot (variable "p")))
 ((register 2 32) (annot (variable "i")))))
(frame_layout
 ((const (symbol "arr")) 64)
 ((const (symbol "buf")) 224))
(flowgraph
 ((0 (node (stmt_enter))
  (1 (node (stmt_def (register 2)
   (expr_imm (const (bits "00000002"))))))
  (2 (node (stmt_store
   (expr_op (operator add 32 (32 32))
    ((expr_rel (const (symbol "buf")))
     (expr_imm (const (bits "00000014")))))
     (expr_use (register 2)))))
  (3 (node (stmt_def (register 1)
   (expr_rel const (symbol "buf")))))
  (4 (node (stmt_def (register 1)
   (expr_op (operator add 32 (32 32))
    ((expr_use (register 1))
    (expr_imm (const (bits "00000004"))))))))
...
```

(c) Corresponding NIC code fragment



(d) Potential memory layout and environment



(e) Abstract memory and environment

Figure 2. C to NIC conversion example (a-d) and corresponding abstract state (e)

closely, such that all data and control structures in the C code have direct counterparts in the NIC code. Then flow analysis results on NIC, such as execution count bounds on basic blocks or iteration bounds on loops, can be mapped back to the corresponding control-flow construct in C. But NIC can also represent the code after structural optimizations, before the backend generates object code. The control flow of this kind of NIC code will then be close to the control flow of the object code, and flow analysis results can be mapped to the object code with only minor modifications.

NIC can be seen as high-level assembler code format and is written in a LISP-like syntax. All data values are represented as bitstrings and the interpretation of a certain bitstring depends on the operation applied to the datum. For example, `SMUL(32,32)` is an operation specified for two 32-bit bitstrings where the arguments should be interpreted as signed 32 bit integers. Few properties of the target architecture are present in NIC. However, sizes of different data types, endianness and alignment have to be specified.

NIC supports three type of places where data can be stored: *global memory* (allocated at program start-up), *local memory* (allocated at function entries) and *registers* (or temporaries). NIC has a common global symbol table including global data identifiers and function identifiers, the latter allowing for function pointers. Data that can be referenced using pointers cannot be allocated in registers, but are instead stored in local memory. The same holds for large local data allocated at function entries, like arrays and structs.

Global and local data are allocated by NIC in *memory frames*. NIC does not specify the actual memory address for a frame, just that the frame should be allocated when entering the function and that it should have a certain size. An allocated frame can be referenced by a given frame id. A hypothetical implementation of NIC would use an *environment* as part of the state, which maps frame id's to actual addresses in such a way that frames don't overlap. See Fig. 2.

All global data frames are allocated and given values at program startup. Local data frames are allocated at function entries and deallocated at function returns. Whenever a new frame is allocated the current environment is updated with a mapping between the frame id (used to reference the frame) and the address where the frame was allocated. This means that a new local frame can be allocated every time a function is called, and the environment makes sure that the last allocated frame is referred by the given frame id.

As an illustrating example consider the C code fragment in Figure 2(b) and corresponding NIC code fragment in Figure 2(c). The `int buf[7];` declaration has a direct correspondance in the NIC code, specifying that a frame of size $32 * 7 = 224$ bits should be allocated and referenced using the frame id "`buf`". The `i` variable and `p` pointer are not allocated in frames but are instead stored in registers. Figure 2(d) gives a potential memory layout after the NIC code has been executed. It also gives the resulting environment.

A pointer value in NIC is a pair (x, o), where x is a frame id and o is an offset. Given an environment E, the actual reference to memory is calculated as $E(x)+o$. For accesses of array elements $a[i]$ the offset is calculated as $s \cdot i$ where s is the size in bytes of the elements, just as in C.

```
   char a[5]={'h','e','l','l','o'};
   char b; char c = 100; char *p;
1) if(b < 10)
     p = &a[2];  2)
   else
     p = &a[0];  3)
   p++;
4) if(b == 5)
     p = &c;     5)
   b = *p;       6)
```

(a) Example C code program

$S^1 = [a \mapsto P, b \mapsto [-128..127], c \mapsto [100..100], p \mapsto \bot]$
$S^2 = [a \mapsto P, b \mapsto [-128..9], c \mapsto [100..100], p \mapsto \{\langle "a", [2..2] \rangle\}]$
$S^3 = [a \mapsto P, b \mapsto [10..127], c \mapsto [100..100], p \mapsto \{\langle "a", [0..0] \rangle\}]$
$S^4 = [a \mapsto P, b \mapsto [-128..127], c \mapsto [100..100], p \mapsto \{\langle "a", [1..3] \rangle\}]$
$S^5 = [a \mapsto P, b \mapsto [5..5], c \mapsto [100..100], p \mapsto \{\langle "c", [0..0] \rangle\}]$
$S^6 = [a \mapsto P, b \mapsto [100..108], c \mapsto [100..100], p \mapsto \{\langle "a", [1..3] \rangle, \langle "c", [0..0] \rangle\}]$
where
$P = \{\langle 0, [104..104] \rangle, \langle 1, [101..101] \rangle, \langle 2, [108..108] \rangle, \langle 3, [108..108] \rangle, \langle 4, [111..111] \rangle\}$

(b) Abstract values

Figure 3. Example program with new abstract domain

NIC has a general function call parameter transfer. The caller pushes evaluated parameter values in a parameter passing area as specified by a specific call instruction, and the called function pops the corresponding parameter values into its local registers. At a function entry a new environment is allocated and updated with new registers and new mappings between frame id:s and frame start addresses. The old environment is stored on a call stack. At function returns, a similar mechanism is used to handle return values from functions. The previous environment is restored from the call stack.

5.3. Our New Analysis Domain

Our new abstract domain, for our flow analysis, is defined for NIC. For ordinary numerical values we use intervals, as outlined in Section 3, to represent sets of values. Each basic data type in C, such as `char`, `short`, `int` and `float` has a corresponding abstract interval data type. The $-\infty$ and $+\infty$ values used in classical interval analysis are replaced with the real lower and upper bounds of each particular data type. The abstract domain of each such abstract type forms a lattice with a specified bottom element and top element, as outlined in Section 3. For example, for an unsigned short the top element $\top_{\text{short}}$ is $[0..65535]$ as shown in Figure 2(e). The bottom element occurs when a data cannot contain any value, which is an indication of an infeasible execution path.

All data are given a default abstract basic type according to its specified C type. Each data can only have one current abstract type, except for integers where we concurrently maintain both a signed and unsigned abstract value domain, see Section 5.4. If the value should be interpreted as another type than its default type, maybe due to some type cast, we have functions for converting data between different domains, e.g., from an integer interval to a float interval. We have also implemented abstract versions of all bit manipulating operations for intervals.

For larger pieces of memory consisting of several basic data, like arrays and structs, we use the NIC frame concept. An abstract frame is defined as a set of abstract data, where each abstract data has a specified abstract type and a specified start address in the frame. For example, in Figure 2(e) the `buf` array gets represented as an abstract frame `frame2` of 224 bits size which contains seven 32-bit integer intervals stored continously at every 32-bit offset from the frame start address.

We also use the NIC frame concept to implement pointer values. An abstract data in an abstract frame is referenced using a symbolic frame id and an offset, where the offset is represented as an integer interval. For example, the possible value of `&a[2];` is represented as a tuple $\langle "a", [2..2] \rangle$, where a is the symbolic frame id "a" used to refer to the frame allocated for array `a`. To handle the case that a pointer can point to several different data, we represent a pointer value as a set of such tuples.

Figure 3 gives an illustration of our pointer representation. Since variable `b` is locally allocated an not initialized with a value we cannot decide the outcome of the different conditionals. A safe approximation of the possible state after executing the code in Figure 3(a) is given by program point S^6 in Figure 3(b). The abstract value of `p` of $\{\langle "a", [1..3] \rangle, \langle "c", [0..0] \rangle\}$ holds all potential pointer values of `p` and is used to give values to the `b` variable.

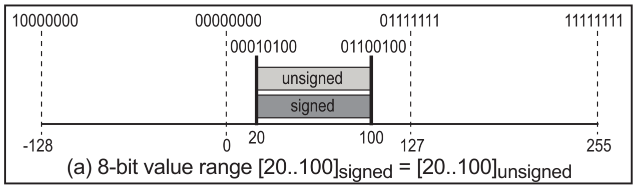
We note that our abstract pointer representation will never include frames which cannot be referenced by the pointer. For example, in Figure 3 the value of `p` will never include `b` even though `b` might be located at an physical address situated inbetween `a` and `c`. This pointer representation also allows us to handle operations like addition and subtraction on pointers, by applying the operation on the offset interval. For example, the `p++` operation in Figure 3(a) increases the offset value by one.

In our current implementation we only keep one offset interval per frame id. This might sometimes yield infeasible offset values. For example, in Figure 3 the $\langle "a", [1..3] \rangle$ is overly pessimistic since `p` actually can only point to the addresses of `a[1]` and `a[3]`. We plan to evaluate the pessimism introduced by this representation, and increase the number of offsets per frame id if neccessary.

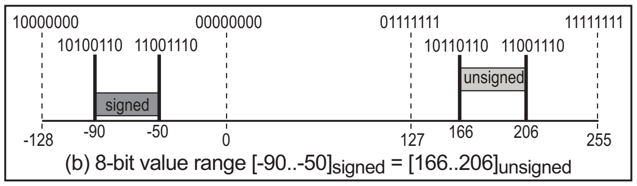
5.4 Representing Both Signed and Unsigned

In classical interval analysis each variable value has a designated type, and is represented as a range consisting of an lower and upper limit of the specified value. For example, a 8-bit unsigned char variable value would be represented as a range $[i..j]$ where $0 \leq i \leq j \leq 255$. However, as mentioned in Section 5.1, in C implicit or explicit type conversions are often made, and the same data can be used in both signed and unsigned operations. This means that the same bitpattern can represent several possible values, depending on the operation applied on the bitpattern. This provides a problem when designing a useful abstract domain for an interval analysis.

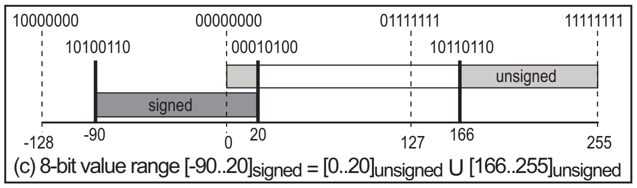
To exemplify the problem, consider an 8-bit character v. If interpreted as unsigned, v can have a value $0 \leq v \leq 255$, and if interpreted as signed, v can have a value $-128 \leq v \leq 127$. This means that some of the possible bitpatterns of v have a different meaning if interpreted as signed respectively unsigned, but not all. For example, bitpattern 00010100 means 20 both as signed and unsigned, while the bitpattern 10100110 is interpreted as -90 if signed and 166 if unsigned. The same principle holds for C:s representation of chars, shorts, ints etc.



(a) 8-bit value range $[20..100]_{signed} = [20..100]_{unsigned}$



(b) 8-bit value range $[-90..-50]_{signed} = [166..206]_{unsigned}$



(c) 8-bit value range $[-90..20]_{signed} = [0..20]_{unsigned} \cup [166..255]_{unsigned}$

Figure 4. Signed and unsigned integer ranges

Figure 4 illustrates the problem of representing values using standard integer intervals. As shown in Figure 4(a), an 8-bit value range $[00010100..01100100]$ is interpreted as the same value range both as signed and as unsigned, $[20..100]$. The 8-bit value range $[10100110..11001110]$ becomes more problematic since it will be interpreted differently as signed, $[-90..-50]$, respectively unsigned, $[166..206]$. This means that a $<$ comparison between the two bitpattern ranges in Figure 4(b) will have different outcomes depending on if a signed or unsigned comparison is applied.

When value ranges cross signed and unsigned bitvalue borders the problem becomes even more problematic. Figure 4(c) illustrates the range $[-90..20]$, and its corresponding bitpattern range: $[10100110..00010100]$. The range includes values which are the same if interpreted as signed or unsigned, but also some values which are not. If we want to interpret the $[-90..20]$ range as an unsigned, e.g., to apply a unsigned operation, we get the *two* unsigned ranges $[0..20]$ and $[166..255]$ as a result, where the latter corresponds to the part of the $[-90..20]$ range smaller than 0.

A safe, but very pessimistic conversion of $[-90..20]$ to an unsigned range would be the $[0..255]$ range, i.e., a range holding all possible unsigned values for an 8-bit value. However, this conversion means that we lose a lot of precision when going from signed to unsigned and vice versa. Consequently, when going back to the signed value representation again, after applying the unsigned operation, the resulting value would be a gross overapproximation, i.e., the range $[-128..127]$. For larger data, like 32-bit integer intervals, a signed to unsigned conversion might easily lead to very large precision loss, resulting in an range representing all possible 32-bit integer values.

To solve this problem, we chose to maintain *two* representations of integers at the same time; one signed, and one unsigned. Each representation can consist of at most two ranges. For a representation to consist of two ranges it must have the property to include both the min and max value limits in the range, and that the corresponding (signed or unsigned) representation only consists of one range. Figure 4(c) illustrates the idea. The signed $[-90..20]$ range corresponds to the two $[0..20]$ and $[166..255]$ unsigned ranges. The unsigned representation is allowed to be two ranges since it includes the min value (0), and max value (255) limits in the range.

This double range representation also nicely handles the problem of value limit overflow and underflow for a range. For example, adding the value of one to an unsigned 8-bit range $[250..255]$, will result in a overflow, yielding the two unsigned ranges $[0..0]$ and $[251..255]$, (together corresponding to the signed range $[-5..0]$), instead of the overly pessimistic range $[0..255]$. The drawback of this representation is that all abstract operations have to be adapted to arguments consisting of two ranges instead of one. For example, the abstract version of `SMUL` must be able to multiply two arguments each consisting of (at most) two ranges.

6. The SWEET Tool

The SWEET tool (SWEdish Execution time Tool) [15, 22] is a research prototype tool, which combines the flow

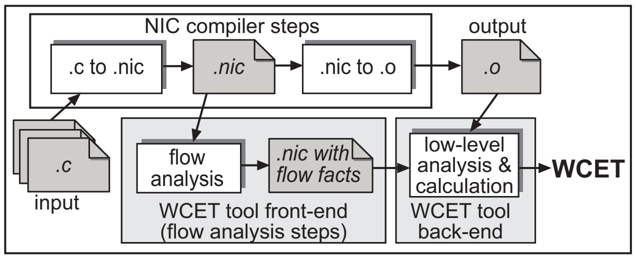


Figure 5. WCET tool prototype

analysis described in this paper with a number of supporting analyses. For example, in order to speed up the analysis, statements not affecting the control flow are removed, and simple loops are analysed and removed prior to the abstract interpretation step.

The basic analysis steps of SWEET are depicted in Figure 5. The C program is translated to NIC code, which is analysed with the flow analysis. The result of the flow analysis is stored as flow facts, which are combined with the result of the low-level analysis (including pipeline analysis) in a calculation step to yield a final WCET.

7. Conclusions and Future Work

We have shown how abstract interpretation can be used to make flow analysis on embedded C code. One important next step will be to test these ideas in practice on real embedded C code. This will be done once the SWEET tool is finished.

We are participating in the ARTIST2 timing analysis cluster together with most WCET analysis researcher in Europe. The aim for the cluster is to increase the availability of static WCET tools and analyses. This is made by the development of common data formats for exchanging analysis results, such as flow facts, required in the WCET analysis.

For future work one idea is to map the flow facts found for (unoptimized) NIC code back to C. In this way, our flow analysis results would be easily visible by the programmer. Also, the flow analysis will be more autonomous and less coupled to our internal data representation. Another idea is to convert assembler code to NIC to be able to perform flow analysis on code for which source code is missing.

Acknowledgements

This research has been supported by the Advanced Software Technology Center (ASTEC) in Uppsala [2]. ASTEC is a Vinnova (Swedish Agency for Innovation Systems) initiative [36].

References

[1] AbsInt. AbsInt company homepage, 2004. URL: `http://www.absint.com`.

[2] ASTEC homepage, 2004. URL: `http://www.astec.uu.se`.

[3] I. Bate and R. Reutemann. Worst-Case Execution Time Analysis for Dynamic Branch Predictors. In *Proc. 16th Euromicro Conference of Real-Time Systems, (ECRTS'04)*, June 2004.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, pages 85–108. Lecture Notes in Computer Science (LNCS) 2566. Springer-Verlag, 2002.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM Press, 2003.

[6] F. Bourdoncle. Assertion-based debugging of imperative programs by abstract interpretation. In *Proceedings of the Fourth European Software Engineering Conference*, 1993.

[7] S. Byhlin. Evaluation of Static Time Analysis for Volcano Communications Technologies AB. Master's thesis, Mälardalen University, Västerås, Sweden, Sept 2004.

[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. 2nd International Workshop on Real-Time Tools (RT-TOOLS'2002)*, 2002.

[9] P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.

[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. 4th ACM Symposium on Principles of Programming Languages*, Los Angeles, January 1977.

[11] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. Lecture Notes in Computer Science (LNCS) 631, pages 269–295. Springer-Verlag, August 1992.

[12] J. Engblom. Why SpecInt95 Should Not Be Used to Benchmark Embedded Systems Tools. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, May 1999.

[13] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Box 337, Uppsala, Sweden, Apr 2002. ISBN 91-554-5228-0.

[14] J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proc. 21th IEEE Real-Time Systems Symposium (RTSS'00)*, Nov 2000.

[15] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Box 325, Uppsala, Sweden, June 2003. ISBN 91-554-5671-5.

[16] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer Verlag, Aug 1997.

[17] 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.

[18] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[19] J. Ganssle. Really Real-Time Systems. In *Proc. of the Embedded Systems Conference San Fransisco 2001*, Apr 2001.

[20] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

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

[22] J. Gustafsson, B. Lisper, C. Sandberg, and N. Bermudo. A Tool for Automatic Flow Analysis of C-programs for WCET Calculation. In *8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, Jan 2003.

[23] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding Loop Iterations for Timing Analysis. In *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, June 1998.

[24] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *IEEE Proceedings on Real-Time Systems*, 2003.

[25] N. Holsti, T. Långbacka, and S. Saarinen. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, Sep 2000.

[26] R. Kirner and P. Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*. IEEE Computer Society Press, June 2001.

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

[28] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997. ISBN: 1-55860-320-4.

[29] Homepage for PolySpace, Nov 2004.
URL: `http://www.polyspace.com/`.

[30] M. Rosendahl. Automatic Complexity Analysis. In *Proceedings of Functional Programming Languages and Computer Architecture Conference, FPCA '89*, pages 144–156. ACM Press, 1989.

[31] C. Sandberg. Inspection of Industrial Code for Syntactical Loop Analysis. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, June 2003.

[32] 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.

[33] V. Seppänen, A-M Kähkönen, M. Oivo, H. Perunka, P. Isomursu, and P. Pulli. Strategic needs and future trends of embedded software. Technical Report Technology Review 48/96, TEKES Technology Development Center, Oulu, Finland, Oct 1996.

[34] D. B. Stewart. Measuring Execution Time and Real-Time Performance. In *Proceedings of the Embedded Systems Conference (ESC SF) 2002*, Mar 2002.

[35] A. Venet and G. Brat. Precise and Efficient Static Array Bound Checking for Large Embedded C Programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, June 2004.

[36] Vinnova homepage, 2004.
URL: `http://www.vinnova.se/`.

[37] The Whole-Program Optimization project homepage, 2001.
URL: `http://www.astec.uu.se/etapp3/`.