Deriving Annotations for Tight Calculation of Execution Time[⋆]

Andreas Ermedahl[1] and Jan Gustafsson[2]

[1] Dept. of Computer Systems, Uppsala University, Sweden, ebbe@docs.uu.se
[2] Dept. of Computer Engineering, Mälardalens högskola, Sweden, jgn@mdh.se

Abstract. A number of methods have been presented to calculate the worst case execution time (WCET) of real-time programs. However, to properly handle semantic dependencies, which in most cases is needed to reduce overestimation, all these methods require extra semantic information to be given by the programmer (manual annotations for paths, loops and recursion depth). To manually derive these annotations is often difficult and the process is error-prone. In this paper we present a new method to automatically derive safe and tight annotations for paths and loops. We illustrate our method by giving some examples and by presenting a prototype tool, implementing the method for a subset of C.

1 Introduction

Real-time systems are systems in which the correctness depends not only on the results of computations, but also on the time at which the result is produced. To be able to guarantee the deadlines of a real-time system, the software execution time is needed before run-time.

The execution time of most programs depends on the input data and the system state. For programs with some complexity, it is intractable to find the data and the state which causes the *actual* worst case execution time (WCET_A). This approach is therefore not a feasible method. Instead, *static analysis*, which from the source code derives the *calculated* worst case execution time (WCET_C), has been proposed by many researchers. The calculation must be *safe* (i.e., $\mathrm{WCET}_C \geq \mathrm{WCET}_A$), yet as *tight* as possible, to avoid waste of resources.

To achieve a tight WCET_C, more information about the program behaviour, than what is contained in the flow graph, is needed. *False paths* (non-executable paths, i.e., paths that never can be taken) must be identified and excluded from the calculation. Maximum number of *iterations* in loops and maximum *depth of recursion calls* must be given, because to calculate them is, in the general case, equivalent to the well-known halting problem.

Existing methods require this information to be given as *manual annotations.* However, a fundamental problem with this approach, beside being difficult and

[⋆] This work is performed within the ART-project within the Advanced Software Technology (ASTEC) competence centre, and is supported by the Swedish board for technical development (NUTEK), IAR Systems AB, Mecel AB and Uppsala University.

time-consuming for the programmer, is that the annotations may be *incorrect*. The WCET_C may be untight or, worse, unsafe.

In this paper we present a static analysis method which *automatically* derives safe and tight annotations from the program semantics. It can be seen as a first phase in a tight WCET_C calculation. These annotations can be used by the following phase, an object code analysis, which also considers modern hardware architectures. If simple hardware is used, this next phase may be unnecessary, and the WCET_C calculated by our method can be used.

The remainder of the paper is organised as follows: The next section illustrates how our approach relates to other methods. In section 3, we introduce our new method. In section 4 we present our tool and an illustrative example. Finally, section 5 gives some results, conclusions and ideas for future work.

2 Related work

Timing analysis of software has been an important area in real-time research during the last 8 - 10 years. The issues this research has dealt with are:

1. Mapping the high-level source code constructions to the corresponding object code instructions by *static analysis*. Compiler optimisations will make this more difficult [19].
2. Deriving *dynamic* properties of programs, i.e., how many times each instruction will be executed in the worst case. Information of *false paths*, maximum number of *iterations* in loops and maximum *depth of recursion calls* are normally given as manual annotations. However, *symbolic execution methods* has been proven to find some false paths automatically [1,4].
3. Calculating the WCET_C using the static and dynamic information above. *Timing schema* [15,16] analysis calculates WCET_C by recursively adding the times for the constructs in the program. Another method is *Integer Linear Programming* (ILP) [9,17] analysis, which is used in most recent research. It transforms the program to a flow graph where each edge corresponds to a basic block. For each edge, the worst execution time is calculated from the instructions in the basic block. The total execution time is represented as a linear expression, and the WCET_C is calculated by finding the maximum of the expression with linear programming optimisation.
4. Modern hardware with cache and pipeline is analysed in extensions to TS [11], ILP [10] or using constraint techniques [13]. This analysis must take into consideration the current cache and pipeline contents before each instruction execution. To get a tight calculation, detailed information on dynamic program behaviour, especially nested loops, is needed.

As we can see, all published methods rely more or less on manual annotations to work and to give a small overestimation. But to give these annotations is extra work for a stressed programmer, and it is certainly easy to find very simple examples where, e.g., the maximum number of iterations in a loop is very hard to calculate (see Fig. 4(a) for an example). And what happens if the manual annotation is incorrect? One possible effect is that a program could be given a too small time slot in a schedule, ending up in a missed deadline, with possible catastrophic consequences.

Park discusses this problem in [14], proposing a method that verifies the correctness of the annotations. But why not let the analysis method try to find the annotations automatically, as they are inherent in the semantics of the program?

3 Deriving Annotations

In our analysis of a program we will use data flow analysis to find out values of variables at different points in the program. Using this information we can automatically derive path and loop annotations.

Control points, $c_0, c_1, \cdots, c_m$, are introduced at points in the program where the value of a variable may change (after assignments), or when we can constrain the possible values of variables (after conditions). For an example, see Fig. 1(a).

With each control point, c_i, we associate an *environment*, σ_i^h. An environment holds all combinations of variables values that are possible at the control point in a program execution. $\sigma_i^h = \{a_1 \mapsto v_1, \cdots, a_m \mapsto v_m\}$ denotes an environment where the variables $a_1, \cdots, a_m$ have been assigned the values $v_1, \cdots, v_m$, respectively. We will by the index h separate between different passings of the same control point, e.g., in loops and continuations after selection statements (h will in the sequel only be included when necessary).

$[c_0]$	σ_0	$\sigma_0 = \{\mathrm{a} \mapsto 1, \mathrm{b} \mapsto 4\}$	$\sigma_0 = \{\mathrm{a} \mapsto 1..3, \mathrm{b} \mapsto 2..4 \vee 7\}$
`a = 5;` $[c_1]$	$\sigma_1 = [\![\texttt{a = 5}]\!]\sigma_0$	$\sigma_1 = \{\mathrm{a} \mapsto 5, \mathrm{b} \mapsto 4\}$	$\sigma_1 = \{\mathrm{a} \mapsto 5, \mathrm{b} \mapsto 2..4 \vee 7\}$
`b = b - 2;` $[c_2]$	$\sigma_2 = [\![\texttt{b = b - 2}]\!]\sigma_1$	$\sigma_2 = \{\mathrm{a} \mapsto 5, \mathrm{b} \mapsto 2\}$	$\sigma_2 = \{\mathrm{a} \mapsto 5, \mathrm{b} \mapsto 0..2 \vee 5\}$
`a = a + b;` $[c_3]$	$\sigma_3 = [\![\texttt{a = a + b}]\!]\sigma_2$	$\sigma_3 = \{\mathrm{a} \mapsto 7, \mathrm{b} \mapsto 2\}$	$\sigma_3 = \{\mathrm{a} \mapsto 5..7 \vee 10, \mathrm{b} \mapsto 0..2 \vee 5\}$
(a)	(b)	(c)	(d)

Fig. 1. The statements (with inserted control points) in (a) gives the semantic rules in (b) and the concrete and abstract evaluation in (c) respectively (d).

3.1 Concrete and abstract semantics

The *concrete semantics* (meaning) of a program is defined as the environments that can be generated by the program description [12]. We will use a semantic function, $[\![\cdot]\!]$, that takes an environment, σ, and a *rule* on how to modify (or constrain) the environment and return a modified environment: $\sigma' = [\![rule]\!]\sigma$. Depending on the nature of the rule, i.e., if it is a statement or a condition, we will further subdivide $[\![\cdot]\!]$ into $\mathcal{S}[\![\cdot]\!]$ and $\mathcal{C}[\![\cdot]\!]$[1]. See Fig. 1(b) for the semantic rules that corresponds to the statements in 1(a). If each variable in the initial environment, σ_0, is assigned to a single value, like $\sigma_0 = \{\texttt{a} \mapsto 1, \texttt{b} \mapsto 4\}$, the evaluation of the equations will correspond to a "normal" execution of the program, see Fig. 1(c).

[1] $\mathcal{S}$ stands for *statement* and $\mathcal{C}$ for *condition* rule.

For our analysis, we will define an *abstract environment* where variables can be assigned to several values. For each concrete semantic rule, in the programming language, a corresponding abstract rule is defined. For example, our abstract version of the '+' operator handles sets of values. An abstract evaluation can be seen in Fig. 1(d). Note that the abstract evaluation corresponds to a set of concrete evaluations and that each concrete evaluation corresponds to a possible execution.

In this paper, and in our tool (see section 4), we will represent abstract values with *split integer intervals*. For example, $\texttt{b} \mapsto 2..4 \vee 7$ means that `b`:s value is either 7 or between 2 and 4. This representation has some drawbacks, e.g., it does not express conditions between variables in an environment, on the other hand it is simple to manipulate and allows efficient implementations.

The domain of the environments is in the general case a partially ordered set (poset), $\langle \mathcal{D}, \subseteq, \perp, \cup, \cap \rangle$, The set $\mathcal{D}$ contains possible combinations of value tuples for variables. The bottom element, $\perp$, means that one or several of the variables within the environment can not have any value at all. $\sigma_1 \subseteq \sigma_2$ is true iff at least all tuples of variable value combinations that exists in σ_1 also exists in σ_2. $\sigma_1 \cup \sigma_2$ creates a new environment that holds exactly all tuples of variable value combinations in *both* σ_1 and σ_2.

The cost, in terms of time and memory, to express an exact environment, i.e., all possible tuples of variables value combinations, is often to expensive. The representation can then be simplified by a safe abstraction, e.g., $a \mapsto 1 \vee 3 \vee .. \vee 99$ can be safely approximated to be in the interval 1..99. The approximation must be safe (possible values must not be removed), tight (as few extra values as possible), and efficient (in terms of time and memory). We will face a trade-off between cost of computation and quality of results. For the above reason we will also use an approximative meet operator $\cup'$ that applied on σ_1 and σ_2 creates a new environment that holds *at least* all tuples of variable value combinations in σ_1 and σ_2. Abstract interpretation techniques [5] can be used to define a correct relation between the abstract and the concrete domains.

3.2 Finding false paths

We will use a sequence of `if`-statements to illustrate how dependencies between different program parts can be found.

A condition can be seen as a constraint to be applied on the variables in a given environment. Fig. 2(c) shows how the start environment, $\sigma_0 = \{\texttt{a} \mapsto 0..20\}$, will be constrained[2] by the conditions in the two `if`-statements in Fig. 2(b). The evaluation in Fig. 2(c) has the disadvantage that it does not take into account the dependencies between the `if`-statements (`a > 10` implies `a > 5`). Thus, it will not detect that $S_1 \rightarrow S_4$ is a false path. Our solution is to continue the analysis from each of the two environments that are generated after an `if`-statement, giving the semantic rule in Fig. 2(a).

[2] We are assuming that `a` will not be changed in any of the statements $S_1 \dots S_4$.

$$\mathcal{S}[\![\texttt{if}\,(C)\ S_1\ \texttt{else}\ S_2]\!]\sigma = \{\sigma_2, \sigma_4\}$$
where
$$\begin{aligned}\sigma_1 &= \mathcal{C}[\![C]\!]\sigma\\ \sigma_2 &= \mathcal{S}[\![S_1]\!]\sigma_1\\ \sigma_3 &= \mathcal{C}[\![\neg C]\!]\sigma\\ \sigma_4 &= \mathcal{S}[\![S_2]\!]\sigma_3\end{aligned}$$
(a)

```
[c0]
if(a > 10) [c1] S1 [c2]
else [c3] S2 [c4]
[c5]
if(a > 5) [c6] S3 [c7]
else [c8] S4 [c9]
[c10]
```
(b)

$$\begin{aligned}\sigma_0 &= \{\mathtt{a} \mapsto 0..20\}\\ \sigma_1 = \sigma_2 &= \{\mathtt{a} \mapsto 11..20\}\\ \sigma_3 = \sigma_4 &= \{\mathtt{a} \mapsto 0..10\}\\ \sigma_5 &= \{\mathtt{a} \mapsto 0..20\}\\ \sigma_6 = \sigma_7 &= \{\mathtt{a} \mapsto 6..20\}\\ \sigma_8 = \sigma_9 &= \{\mathtt{a} \mapsto 0..5\}\\ \sigma_{10} &= \{\mathtt{a} \mapsto 0..20\}\end{aligned}$$
(c)

$$\begin{tikzcd}[column sep=tiny]
& & & \begin{array}{r}\sigma_0 = \{\mathtt{a} \mapsto 0..20\}\\ \sigma_1 = \sigma_2 = \{\mathtt{a} \mapsto 11..20\}\\ \sigma_3 = \sigma_4 = \{\mathtt{a} \mapsto 0..10\}\end{array} \arrow[dll] \arrow[drr] & & & \\
& \begin{array}{r}\sigma_5^{c_2} = \{\mathtt{a} \mapsto 11..20\}\\ \sigma_6^{c_2} = \sigma_7^{c_2} = \{\mathtt{a} \mapsto 11..20\}\\ \sigma_8^{c_2} = \sigma_9^{c_2} = \{\mathtt{a} \mapsto \perp\}\end{array} \arrow[dl] \arrow[dr] & & & & \begin{array}{r}\sigma_5^{c_4} = \{\mathtt{a} \mapsto 0..10\}\\ \sigma_6^{c_4} = \sigma_8^{c_4} = \{\mathtt{a} \mapsto 6..10\}\\ \sigma_8^{c_4} = \sigma_9^{c_4} = \{\mathtt{a} \mapsto 0..5\}\end{array} \arrow[dl] \arrow[dr] & \\
\sigma_{10}^{c_2,c_7} = \{\mathtt{a} \mapsto 11..20\} & & \sigma_{10}^{c_2,c_9} = \{\mathtt{a} \mapsto \perp\} & & \sigma_{10}^{c_4,c_7} = \{\mathtt{a} \mapsto 6..10\} & & \sigma_{10}^{c_4,c_9} = \{\mathtt{a} \mapsto 0..5\}
\end{tikzcd}$$
(d)

Fig. 2. The semantic rule for an if-statement in (a) gives for the program in (b) the evaluation in (d) instead of the one in (c).

As seen in Fig. 2(d) each if-statement will now generate *two* different environments. It can now be seen that the path `S1` $\rightarrow$ `S4` is a false path.

3.3 Finding the number of iterations in loops

We will use a `while`-statement to illustrate how our analysis method will work for loops. The core idea is to transform them into `if`-statements, giving the semantic rule in Fig. 3(a). The `if`-statement yields two environments each time it is analysed (as before):
1. The environment in which the loop shall be executed again, σ_{true}.
2. The environment in which the loop terminates, σ_{false}.

$$\mathcal{S}[\![\texttt{while}(C)\ S]\!]\sigma = \mathcal{S}[\![\texttt{if}(C)\{S;\texttt{while}(C)\ S\}]\!]\sigma$$
(a)

```
while (a < 9) { [c_true]
   a = a + 2;
   } [c_false]
```
(b)

Iter	σ_{true}	σ_{false}	
0	$\{\mathtt{a} \mapsto 0..2 \vee 5\}$	$\{\mathtt{a} \mapsto \perp\}$	
1	$\{\mathtt{a} \mapsto 2..4 \vee 7\}$	$\{\mathtt{a} \mapsto \perp\}$	
2	$\{\mathtt{a} \mapsto 4..6\}$	$\{\mathtt{a} \mapsto 9\}$	min = 2
3	$\{\mathtt{a} \mapsto 6..8\}$	$\{\mathtt{a} \mapsto \perp\}$	
4	$\{\mathtt{a} \mapsto 8\}$	$\{\mathtt{a} \mapsto 9..10\}$	
5	$\{\mathtt{a} \mapsto \perp\}$	$\{\mathtt{a} \mapsto 10\}$	max = 5

(c)

Fig. 3. The semantic rule for loop-evaluation in (a) gives for the program in (b) the table in (c).

A loop is "rolled out" until it cannot execute again, or until the time budget is exceeded (see section 3.5). For example, with the start environment $\sigma = \{\mathtt{a} \mapsto 0..2 \vee 5\}$, the code in Fig. 3(a) will generate the table in Fig. 3(b). The analysis shows that the loop will iterate at least two times (since 2 iterations is needed to set $\sigma_{false} \neq \bot$) and at most 5 times (since after 5 iterations $\sigma_{true} = \bot$, which means that we cannot enter the loop again). The analysis also shows that `a` always will be in the interval 9..10 after the loop.

3.4 Merging environments

Environments will, for several reasons, be merged (using the $\cup$ or $\cup'$ operations) at certain points during the analysis. In our current tool the chosen merge points are the ends of loops, functions and programs. The reasons for merging are:

1. Many evaluations from σ_{false}- environments will be redundant. For example, $\sigma^{h_i}_{false} \subseteq \sigma^{h_j}_{false}$, means that $\sigma^{h_i}_{false}$ is redundant since the evaluation from $\sigma^{h_j}_{false}$ will include all possible executions that could result from $\sigma^{h_i}_{false}$[3].
2. To reduce the number of continuous evaluations. For example, a loop body with n if-statements will generate 2^n environments for each iteration. Merging of (possible) non-redundant environments will reduce the computational cost. However, overestimation may occur.
3. The goal for the analysis of a program is to generate annotations for the corresponding flow graph. Several methods for low level cache- and pipeline-analysis demands this [8,10,11,13]. The annotations must then be true for *all* iterations of each loop.

3.5 Introducing time

The analysis described so far will often not terminate if the program does not terminate. To terminate our analysis, we will use the fact that a real-time program must complete its task within a given deadline. A program is given a time budget, T_{budget}, which should be a realistic upper time limit for the program on a given hardware. The time budget may be calculated during the design phase and can be seen as part of the specification of the program. The T_{budget} is the only manual "annotation" needed by our method[4].

Each statement or program block has a minimum and a maximum execution time, t_{minc} and t_{maxc}. For each analysed statement the corresponding time interval will be added to a accumulated time. The time for the longest path, $T_{minc}..T_{maxc}$, will be compared to the time budget during the analysis. Three cases can be identified:

[3] In Fig. 3(b) the continuing analysis from σ_{false} in both the second and fifth iteration can be included in the continuing analysis from σ_{false} in the fourth iteration.

[4] Note that our time annotation is different from the annotations of other methods. Erroneous path or loop annotations may lead to wrong WCET$_C$, but an erroneous T_{budget} may in the worst case only lead to too early ending of the analysis.

1. $T_{minc} < T_{maxc} < T_{budget}$: In this case we can guarantee that the program will not exceed its time budget.
2. $T_{minc} < T_{budget} < T_{maxc}$: There is now a risk that the program does not terminate within the time budget. Our analysis tool (see section 4) will stop and generate a warning message. If we suspect that the time budget is too narrow, we may extend it and continue further.
3. $T_{budget} < T_{minc} < T_{maxc}$: In this case we know that the program will exceed its time budget. The analysis will normally not reach this point.

Thus, our method calculates WCET_C for the program. However, note that the main reason for this calculation is not to get the WCET_C for the program, but to make sure that the analysis terminates.

4 Implementation and Example

To test the described ideas a prototype tool has been implemented in the programming language Erlang [2]. The tool uses a split integer interval representation of environments. So far only a subset of C is handled, including integer variables and the standard arithmetical operations (`+, -, *, /`), declarations, assignments, selection statements (`if`- and `if-else`- statements) and loop-constructs (`while`- and `for`-statements). Still, this simple language serves to illustrate our ideas. To add more types (e.g., floats), more complicated constructions, (e.g., arrays and structs) will be relatively simple. Functions calls, dynamic memory and pointers demands a much more complicated analysis.

There is also an option in our tool to annotate the code manually with possible input values. This can be used by the programmer to, for instance, study the program behaviour for different inputs.

4.1 Example

The information retrieved from the analysis of the program in Fig. 4(a) is presented in Fig. 4(b) and (c). We are assuming that both `a` and `b` are within the interval 1..30 at the beginning of the program, that is: $\sigma_0 = \{\texttt{a} \mapsto 1..30, \texttt{b} \mapsto 1..30\}$. The values presented in Fig. 4(d) and (e) has been extracted by running the program for all its possible combinations of input values, in this case: $30 * 30 = 900$ executions[5]. For a program with large number of arguments, with varying input values, this is not a feasible option. Our analysis tool extracted, among other things, the following information:

– A safe estimation of the minimum and maximum number of iterations in the outer loop, Fig. 4(a).
– A safe estimation of the minimum and maximum number of iterations in the inner loop for each iteration of the outer loop, Fig. 4(c).
– The fact that the `a = a + 10;` statement never will be executed and therefore is dead code.

[5] Without any abstractions the analysis would derive these values as well.

– A safe estimation of the possible values for a and b within the program, (1..41 and −9..87 respectively).

An interesting comparison can be made between the actual maximum number of iterations in the inner loop: 13, (given by $\sigma_0 = \{a \mapsto 1, b \mapsto 7\}$), the number derived without abstractions: 40, the number derived with our tool: 66, and a coarse manual annotation given by a programmer[6], which could be 300. The reason for the difference, between the actual maximum number of iterations in the inner loop and the values given by the analysis tool, is that the analysis result includes *all* possible executions and reduces the computational cost by using abstractions. When the program iterates very differently depending on the input values our analysis results will of course deteriorate.

```
while(a < 30)
[l_out]
{
  while(b < a)
  [l_in]
  {
    if(b > 5)
      b = b * 3;
    else
      b = b + 2;
    if(b >= 10 &&
       b <= 12)
      a = a + 10;
    else
      a = a + 1;
  }
  a = a + 2;
  b = b - 10;
}
```

(a)

min l_{out}	max l_{out}
0	15

(b)

# l_{out}	min l_{in}	max l_{in}
1	0	5
2	0	10
3	0	9
4	0	8
5	0	7
6	0	6
7	0	5
8	0	4
9	0	3
10	0	2
11	0	2
12	0	1
13	0	1
14	0	1
15	0	1
$\sum$	0	66

(c)

min l_{out}	max l_{out}
0	11

(d)

# l_{out}	min l_{in}	max l_{in}	values giving max # of iterations
1	0	5	$\{a \mapsto 18, b \mapsto 1\}$
2	0	9	$\{a \mapsto 1, b \mapsto 1\}$
3	0	8	$\{a \mapsto 1, b \mapsto 13\}$
4	0	7	$\{a \mapsto 1, b \mapsto 25\}$
5	0	4	$\{a \mapsto 2, b \mapsto 23\}$
6	0	2	$\{a \mapsto 2, b \mapsto 22\}$
7	0	1	$\{a \mapsto 1, b \mapsto 1\}$
8	0	1	$\{a \mapsto 1, b \mapsto 9\}$
9	0	1	$\{a \mapsto 1, b \mapsto 2\}$
10	0	1	$\{a \mapsto 1, b \mapsto 5\}$
11	0	1	$\{a \mapsto 1, b \mapsto 30\}$
$\sum$	0	40	

(e)

Fig. 4. For the program in (a) our tool gives the estimated min and max iterations in the outer (b) respectively inner (c) loop. The actual values are those in (d) and (e).

[6] A programmer may see that a increases with at least 2 every outer iteration, giving $30/2 = 15$ iterations in the outer loop. He may also note that b increases with at least 2 for each iteration in the inner loop. As b also is decreased with 10 for each iteration in the outer loop, the maximum number of iterations in the inner loop will be: $(30 + 10)/2 = 20$ times. This gives a total of $15 * 20 = 300$ iterations.

5 Conclusions, results and future work

We have presented a static analysis method which automatically derives safe and tight annotations from the semantics of the source-code program. Normally, these annotations are given manually, but to derive them is often difficult and error-prone. The analysis shall be seen as a first phase in a tight worst case execution time (WCET$_C$) calculation. The derived annotations can be used by the following phase, an object (micro- or assembler) code analysis, which also considers modern hardware architectures.

A short summary of the information derivable with our method are:

- Information of false paths and dead code within programs (section 3.2).
- Safe estimations of maximum and minimum of iterations both for single and nested loops (section 3.3).
- Possible values for all variables in each point[7], program block or entire program[8] [9].
- A WCET$_C$ that can be used on simple hardware architectures (section 3.5).

As future work we plan to investigate other forms of environment representation. General constraint techniques is one of the candidates [18].

We also plan to investigate how the degree of merging affects the analysis result. We can in one extreme analyse all paths, without merging, but such an analysis will be both very time- and space-consuming. On the other hand, too much merging will generate a lot of pessimism in the analysis.

Backward analysis [6] can be of interest to further enhance our analysis. It is performed by analysing a program backwards from the goal environments.

An obvious future task is to extend the analysed language. An interesting extension will be functions, since a function may have different possible input values at different invocations. These input values can be derived automatically through our method and may lead to a tighter WCET$_C$[10].

Future work will also be to investigate *how* programmers of hard real-time systems are writing their programs. Is there a need for complicated constructions? Should the programmer be forced to write his programs in a certain way, to allow analysis? Can we abandon recursion in real-time programs? An investigation of the programming style used in real-time companies will be performed during spring 1997 to give answers to these and similar questions [7].

References

1. P. Altenbernd. On the false path problem in hard real-time programs. In *Proceedings of the Eight Euromicro Workshop on Real-Time Systems*, pages 102–107, June 1996.

[7] Derived by merging all n environment generated at the i:th control point: $\bigcup_{j=1..n} \sigma_i^{h_j}$.

[8] Derived by merging all environments generated at all different control points.

[9] This information can be used for compiler optimisations (e.g., reduction of size of variables) and program verification (e.g., index checking) [3].

[10] The only other method we know that considers input values for WCET$_C$ calculation is [4], but it relies on manual annotations.

2. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in Erlang.* Prentice Hall, 2 edition, 1996. ISBN 0-13-508301-X.
3. F. Bourdoncle. Abstract debugging of high-order imperative languages. In *Proceedings of SIGPLAN'93 Conference on Programming Language design and Implementation*, pages 46–55, 1993.
4. R. Chapman, A. Burns, and A. Wellings. Integrated program proof and worst-case timing analysis of SPARK Ada. In *ACM Sigplan Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.
6. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming, Proceedings of the Fourth International Symposium, PLILP'92*, volume Lecture Notes in Computer Science 631, pages 269–295, Aug 1992.
7. A. Ermedahl and J. Gustavsson. Real-time industry inquiry of execution time analysis tools. Technical report, Department of Computer Systems, Uppsala University, Sweden, 1997. To be published.
8. M. Harmon, T. Baker, and D. Whalley. A retargetable tecnique for predicting execution time of code segments. *The Journal of Real-Time Systems*, 7, 1994.
9. Y.-T. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM Workshop on Lang., Comp. and Tools for RTS*, May 1995.
10. Y.-T. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *17th IEEE Real-Time Systems Symposium, RTSS'96*, pages 254 – 263, 1996.
11. S. Lim, Y. Bae, G. Jang, B.-D. Rhee, S. Min, C. Park, H. Shin, K.Park, S.-M. Moon, and C. Kim. An accurate worst case timing analysis for risc processors. *IEEE Trans. on Software Engineering*, 21(7):593 – 604, July 1995.
12. H. R. Nielson and F. Nielson. *Semantics with Applications.* John Wiley & Sons, 1992.
13. G. Ottosson and M. Sjödin. Worst-case execution time analysis for modern hardware architectures. In *Proc. SIGPLAN 1997 Workshop on Languages, Compilers and Tools for Real-Time Systems*, June 1997. To appear.
14. C. Park. Predicting program execution times by analyzing static and dynamic program paths. *The Journal of Real-Time System*, 5:31–62, 1993.
15. C. Park and A. Shaw. Experiments with a program timing tool based on a source-level timing schema. *Proceeding of 11th IEEE Real-Time Systems Symposium*, pages 72–81, Dec 1990.
16. P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(2):159–176, Sep 1989.
17. P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Report, Techn. Univ., Inst. für Technische Informatik, Vienna, April 1995.
18. E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.
19. A. Vrchoticky. *The Basis for Static Execution Time Prediction.* PhD thesis, Institut für Technische Informatik, Technische Universität Wien, Austria, April 1994.