Algorithms for Infeasible Path Calculation

Jan Gustafsson, Andreas Ermedahl, and Björn Lisper
Department of Computer Science and Electronics, Mälardalen University
Box 883, S-721 23 Västerås, Sweden
{jan.gustafsson, andreas.ermedahl, bjorn.lisper}@mdh.se

Abstract

Static Worst-Case Execution Time (WCET) analysis is a technique to derive upper bounds for the execution times of programs. Such bounds are crucial when designing and verifying real-time systems. A key component in static WCET analysis is to derive flow information, such as loop bounds and infeasible paths.

Such flow information can be provided as either as annotations by the user, can be automatically calculated by a flow analysis, or by a combination of both. To make the analysis as simple, automatic and safe as possible, this flow information should be calculated automatically with no or very limited user interaction.

In this paper we present three novel algorithms to calculate infeasible paths. The algorithms are all designed to be simple and efficient, both in terms of generated flow facts and in analysis running time. The algorithms have been implemented and tested for a set of WCET benchmarks programs.

1 Introduction

To give timing guarantees for embedded and real-time systems, a key parameter is the *worst-case execution time* (WCET) of the software. A *static WCET analysis* analysis finds an upper bound to the WCET of a program, relying on mathematical models of the software and hardware involved. Given that the models are correct, the analysis will derive a timing estimate that is safe, i.e., greater than or equal to the WCET.

To statically derive a timing bound for a program, information on both the *hardware timing characteristics*, such as the execution time of individual instructions, as well as the program's *possible execution flows*, to bound the number of times the instructions can be executed, needs to be derived. The latter includes information about the maximum number of times loops are iterated, which paths through the program that are feasible, execution frequencies of code parts, etc.

This research has been supported by the KK-foundation through grant 2005/0271.

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, since upper bounds on the number of loop iterations must be known in order to derive WCET estimates [8].

Flow analysis can also identify *infeasible paths*, i.e., paths which are executable according to the control-flow graph structure, but not feasible when considering the semantics of the program and the possible inputs. Information on infeasible paths is not necessary to find a WCET estimate, but may tighten it.

This article presents ongoing work to automatically calculate infeasible paths. Three new and complementary algorithms are presented. They have been implemented in our prototype WCET analysis tool and tested for a set of WCET benchmarks programs.

The concrete contributions of this article are:

- We present ongoing work to extend our flow analysis method, called *abstract execution*, to calculate information about infeasible paths.
- We present three algorithms, calculating different types of infeasible path information, allowing us to trade analysis time for flow information precision.
- We show how to make our infeasible path algorithms input data dependent, allowing us to calculate more precise flow information for a program with limitations on its possible input data values.
- We evaluate the effect of our different infeasible path detection algorithms, including the type and amount of flow information generated.

The rest of this paper is organized as follows: In Section 2, we discuss causes of infeasible paths and describe related work. In Section 3, we describe our research prototype, SWEET. Section 4 describes the different algorithms, and Section 5 presents an illustrating example. Section 6 presents analysis results, and in Section 7 we draw some conclusions and discuss future work.

ECRTS 2006
6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2006/667

2 Causes of Infeasible Paths and Related Work

There are two different causes to infeasible paths. The first cause is semantic dependencies that always hold, as illustrated by the following code fragment:

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

Here, both true-branches for the `if` statements are always in conflict[1], and the corresponding path **A-C** can never be taken.

A second cause to infeasible paths is due to limitations of input data values. Such limitations can be used to further limit the set of feasible paths. For example, if we know that `x>5` when the above code is executed, then we can conclude that the paths **A-C**, **A-D**, and **B-D** are all infeasible i.e., we find more infeasible paths with this additional knowledge.

Recent industrial WCET case-studies [7, 9, 17], have shown that it is important to develop good support for both loop bound analysis and infeasible path detection, thereby reducing the need for manual annotations. The case studies also showed that a mode- (giving a WCET estimate under certain system conditions) and input-sensitive WCET analysis often was preferable, in order to obtain better resource utilization and provide a better understanding of the system's timing characteristics. Thus, it should be important to develop input-sensitive infeasible path analyses.

There has been some work on automatic detection of infeasible paths for WCET analysis. Altenbernd [2] uses a combination of path enumeration, path pruning, and symbolic evaluation to find infeasible paths. Kountouris [13] studies detection of infeasible paths in the synchronous real-time language SIGNAL. Liu et al. [14] use symbolic evaluation of higher languages to avoid infeasible paths. Lundqvist and Stenström [15] find loop bounds and infeasible paths by symbolic simulation on the binary code. Healy et al. use value-dependent constraints to find infeasible paths [12]. Aljifri et al. [1] generate only the feasible paths using the concept of partially-known variables. Chen et al. [4] proposed a method that finds infeasible paths by identifying conflicts between assignments and branches, and between different branches.

The proposed infeasible path detection algorithms all use our flow analysis method *abstract execution* [10, 11], which is briefly described in the next section. This method has some similarities with the one of Lundqvist and Stenström [15], as well as with trace partitioning [3]. However, abstract execution uses a more detailed value domain, and it is based on an abstract interpretation framework.

[1] We assume, for simplicity, that the value of `x` is not modified in **A** and **B**.

3 SWEET and Abstract Execution

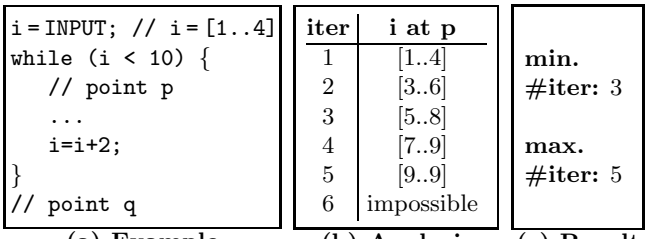
SWEET (SWEdish Execution time Tool) is a prototype WCET tool developed at Uppsala and Mälardalen University [16]. It consists of three main parts; a flow analysis which detects program flow constraints, a low-level analysis, where timing for program parts are obtained [6], and a final calculation where the longest execution path is extracted given information derived in the two preceding stages [8].

The current flow analysis of SWEET uses a *scope-graph* [8]. Each *scope* in the scope-graph is a different execution environment of a program, such as a function or a loop. See Figure 3 for an example. Our current scope-graph representation is context-sensitive, i.e., each call to a function or a loop in a function generates a different scope. Different calls to a function are analysed separately, which may yield higher precision but also a costlier analysis.

Abstract execution is a form of symbolic execution [10, 11], which is based on abstract interpretation. Rather than using traditional fixed-point iteration [5], abstract execution executes the program in the abstract domain, with abstract values for the program variables, and abstract versions of the operators in the language. For instance, the abstract domain can be the domain of intervals: each numeric variable will then hold an interval rather than a number, and each assignment will calculate a new interval from the current intervals held by the variables. As usual in abstract interpretation, the abstract value held by a variable, at some point, represents a set containing the actual concrete values that the variable can hold at that point.

With abstract values, conditionals cannot always be decided, and the abstract execution must then execute both branches. In order to curb the growing number of paths, *merging* of abstract values for different paths can take place. A merged abstract value then surely contains all the possible concrete values from both paths, and a single-path abstract execution, representing the execution of both paths, can continue from the merging point. Typical merge points are places where different program flows meet, like after if-statements or loops. Merging may yield abstract values that represent the possible set of concrete values in a less precise way: for instance, the merge of the intervals [6..6] and [10..11] is [6..11], which also contains the concrete values 7, 8, 9 not present in the original intervals.

SWEET currently supports abstract execution with intervals. It allows the user to control the placement of merge points, in order to explore different tradeoffs



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

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

min.
#iter: 3

max.
#iter: 5

(a) Example **(b) Analysis** **(c) Result**

Figure 1. Example of abstract execution

between analysis speed and precision. Currently, the user can specify merge points to be one or more of the following types: after if-statements, after loop bodies, after loop exits and after function exits.

Figure 1 gives a simple example of abstract execution with intervals. The loop in Figure 1(a) is abstractly executed in Figure 1(b). As iteration 4 and 5 are executed, the set of possible values of `i` is reduced until, finally, the set of values for the true branch of the loop condition is empty, the loop condition is evaluated to FALSE only, and the abstract execution of the loop terminates. During the abstract execution, we keep track of the iteration count of the loop body, and Figure 1(c) shows the resulting loop bounds.

The abstract interpretation framework guarantees that a calculated abstract value always represents the set of possible concrete values. Thus, no execution paths will be missed by the analysis. On the other hand, an abstract value may overestimate this set, which means that the analysis may yield program flow constraints that are not tight. This means that some infeasible paths might be reported as feasible. However, this is safe, since less information about infeasible paths only gives a possibly less tight WCET estimate.

We have created an abstract analysis domain for the data representation in C [11]. This allows us to handle C features like structs, arrays, pointers and type casts. 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 infeasible path analyses presented in this paper are implemented in SWEET as a part of the abstract execution. The abstract execution is input data sensitive, as illustrated in Figure 1, allowing the user to constrain the possible input data values. The result of the abstract execution is passed as flow information, *flow facts* [8], to the subsequent calculation phase. Flow facts are a kind of constraints on the execution count.

4 Algorithms for Infeasible Paths

We now present our three algorithms for infeasible path detection. Since they are based on abstract execution, which is input-sensitive, the analyses are input-sensitive as well.

All three algoritms have a similar overall structure. They augment each analysis state with a *recorder* keeping track of nodes and path(s) taken during a particular analysis of a scope. Each algorithm resets the recorder of a scope when starting a new iteration of the scope. They also associate a *collector* to each scope, which accumulates information about nodes and paths during iterations of the scope. In the end, each collector is used to generate flow facts for its scope.

4.1 Detecting Infeasible Nodes

The first algorithm finds infeasible nodes, that is: basic blocks which are never visited in any execution of a certain scope. Since there is one scope per context, the resulting flow information becomes context sensitive. An infeasible node is therefore not necessarily the same as dead code, since the basic block potentially can be executed in another context.

The recorder object is a bit array with one bit per node in the scope. These bits are all reset to zero at each iteration of the scope, and the bit of a node is set to one at each abstract execution of the node. Thus, a value of zero, after an iteration, means "definitely not executed in this iteration" and one means "may have been executed in this iteration".

The collector object is a similar bit array. Its bits are all initialized to zero, and the end of each iteration the new value of the collector object is set to the bitwise or of its old value and the current value of the recorder object. At termination, if the collector holds a zero for a node, then it is surely never executed in that scope, and a corresponding "infeasible node flow fact" can be generated. An example is:

```
scope : <> : #BB82 = 0;
```

specifying that basic block `BB82` is not executed in any iteration of the scope `scope`.

4.2 Detecting Infeasible Pair of Nodes

The second algorithm finds infeasible pairs of basic blocks, i.e., blocks which are always excluding each other during the same iteration of a scope. This gives additional knowledge as compared to the first analysis, since there might be nodes which both can be executed during some iteration of a scope, but which never can be executed together. The limitation is that infeasible paths with more than two selections can be missed.

The recorder object for this algorithm is a path (list of nodes) taken during an iteration. To limit the number of recorded nodes, only nodes after conditional branches are recorded. At the entry of a scope or at a new loop iteration, the path is emptied. Whenever a conditional branch is taken, we remember the branch

by appending the corresponding node to end of the path. If both paths are taken, the analysis proceeds in two abstract states, one for each path.

The collector object is a triangular matrix which holds exclusion data. It is of size $N \times N$, where N is the number of possible branch outcomes (basic blocks) for the selections in the scope. The matrix can be triangular since the order of the elements in a pair is irrelevant. All elements in such a matrix are set to $\perp$ in the beginning of the analysis, which means that no information is available to start with. A recorder list RL is added to the collector matrix M when an abstract state has reached the end of a loop body or a function scope. The collector is updated as follows:

```
for each node n1 in RL do
  for each subsequent node n2 to n1 in RL do
     M[n1, n2] := 1
  for each alternative branch node n3 to n2 do
     if M[n1, n3] = ⊥ then
        M[n1, n3] := 0
     else
        M[n1, n3] := M[n1, n3] OR 0
```

For example, if the path **A-C** was taken in the example in Section 2 we would have updated M[**A**][**C**] to 1 and M[**A**][**D**] with 0.

When the analysis has finished, the resulting collector matrix is investigated. Matrix positions with $\perp$ mark pairs which have not been touched during the analysis. Some of them can never be executed together anyway due to the structure of the control graph, while the rest really are infeasible pairs. For the first type, generating flow facts will be superfluous. They could be identified using a reachability analysis. However, this is not included in the current implementation, so to avoid a large number of superfluous flow facts, no flow facts are currently generated for $\perp$ positions.

If the matrix positions holds a 0, it marks a node pair that we surely know excludes each other for any iteration of the scope, so for this pair an "excluding pair flow fact" can be generated, like:

```
scope : <> : (#BB33 + #BB57) < 2;
```

specifying that for any iteration of `scope` the basic blocks `BB33` and `BB57` are never executed together.

4.3 Detecting Infeasible Paths

The third algorithm finds sequences of nodes which are never executed together during the same iteration of a scope. The algorithm makes use of the fact that many infeasible paths can be efficiently represented by allowing them to share a common prefix (sub)path.

The recorder data object is now a tree where each tree node represents a path and has an associated

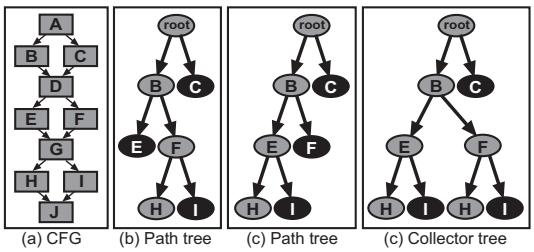


Figure 2. Example CFG and Path Trees

boolean specifying if the corresponding path is feasible or unfeasible. Similar to the recording in the second algorithm we only keep track of nodes taken after branches. However, the tree additionally keeps track of branch outcomes not taken.

Figure 2 gives an illustration of how the recorder tree works. Figure 2(a) gives a CFG with $2^3 = 8$ structurally possible execution paths. Figure 2(b) gives the tree resulting from an execution taking the path **A-B-D-F-G-H-J** through the CFG. In the recorded tree the paths **A-C**, **A-B-D-E** and **A-B-D-F-G-I** have been marked as infeasible. Note that we, for the sake of efficiency, do not record any join nodes in the tree. Similarly, Figure 2(c) gives the tree resulting from an execution path of **A-B-D-E-G-H-J**. Note that the path **A-C** actually represents $2^2 = 4$ number of paths through the CFG, since**A-C** is a prefix of all these paths.

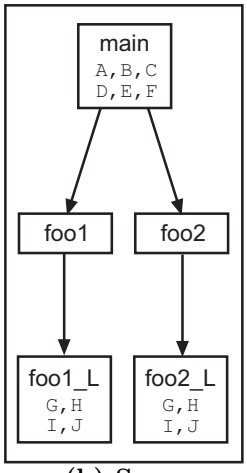
For this algorithm the collector is the tree of paths obtained by merging all recorded trees for the scope. The basic idea of the collector is the same as in the first two algorithms, i.e., only keep infeasible path information which are true for all executions of the scope.

Figure 2(d) gives the collector tree resulting from merging the two trees in Figure 2(b) and Figure 2(c). Both trees has **A-C** as infeasible, and so does the collector tree. Since **A-B-D-E** is infeasible in Figure 2(b) but not in Figure 2(c), since it is part of the feasible **A-B-D-E-G-H-J** path, the collector tree cannot keep the path as infeasible. Instead, all paths (both infeasible and feasible) starting with the path **A-B-D-E** in Figure 2(c) are added to the collector tree. The infeasible path **A-B-D-F** in Figure 2(c) is extended similarly. The resulting collector tree in Figure 2(d) marks paths **A-C**, **A-B-D-F-G-I** and **A-B-D-E-G-I** as infeasible.

A collector tree CT is updated with a recorder tree RT as follows:

```
if RT is the first recorded tree reported
  CT := RT
else
  for each infeasible path i in CT do
    if i is prefix to a feasible path in RT
      mark path i as feasible in CT
      add all paths with prefix i in RT to CT
```

```
x=INPUT; // x=[0..100]
int main(void) {
   if (x<10) ...;      // A
   else ...;           // B
   if (x<5) ...;       // C
   else ...;           // D
   foo(x);
   if (x<0) ...;       // E
   else ...;           // F
   foo(x+50);
   return 1;
}
int foo(int y) {
   int i;
   for (i=0; i<10; i++) {
      if (y>=50) ...; // G
      else ...;       // H
      if (y<50) ...;  // I
      else ...;       // J
   }
}
```



(a) Example code **(b) Scopes**

Figure 3. Code with several infeasible paths

After the analysis we create flow facts for the remaining infeasible paths in the collector tree. An example of such a flow fact is:

```
scope : <> : (#BB33 + #BB57 + #BB82) < 3;
```

specifying that basic blocks BB33, BB57, and BB82 are never executed together for each iteration of `scope`.

5 Example

The example code in Figure 3(a) contains infeasible paths of several types (we assume that neither `i`, `x` or `y` are changed in the excluded code). It will be used to illustrate the algorithms we propose in the paper. The program contains five scopes; `main`, `foo1`, `foo2` (the two calls to `foo`) and the corresponding loop scopes (`foo1_L` and `foo2_L`) in `foo`, as depicted in Figure 3(b). We can identify the following infeasible nodes, pairs and paths:

1. **Infeasible nodes:**
   - **E** is an infeasible node in `main`, **H** and **I** are infeasible nodes in `foo2_L` (limitations in input data).
2. **Infeasible pairs:**
   - **B-C**, **B-E**, and **D-E** are infeasible pairs in `main` (contradicting conditions).
   - **A-E** and **C-E** is an infeasible pair in `main` (limitations in input data).
   - **G-I** and **H-J** are infeasible pairs in `foo1_L` and `foo2_L` (contradicting conditions).
   - **H-I** is an infeasible pair in `foo2_L` (limitations in input data).
3. **Infeasible paths:**
   - **A-D-E**, **B-C-E**, **B-C-F**, and **B-D-E** are infeasible paths in `main` (contradicting conditions).
   - **B-D-E** is an infeasible path in `main` (limitations in input data).

We note that infeasibility can be expressed in several ways, e.g., the infeasible pair **B-C** and the infeasible paths **B-C-E** and **B-C-F** exclude the same paths.

6 Evaluation

Program	Description	#LC	#S	#L
adpcm	Adaptive pulse code modulation algorithm.	879	65	27
bs	Binary search for the array of 15 integer elements.	114	3	1
bsort100	Bubblesort program.	128	4	2
cnt	Counts non-negative numbers in a matrix.	267	10	4
compress	Compression using lzw.	508	22	11
cover	Program for testing many paths.	640	7	3
crc	Cyclic redundancy check computation on 40 bytes of data.	128	11	6
duff	Using "Duff's device" to copy 43 byte array.	86	5	2
edn	Finite Impulse Response (FIR) filter calculations.	285	21	2
expint	Series expansion for computing an exponential integral function	157	5	3
fdct	Fast Discrete Cosine Transform.	239	4	2
fft1	1024-point Fast Fourier Transform using the Cooly-Turkey algorithm.	219	52	30
fibcall	Iterative Fibonacci, used to calculate fib(30).	72	3	1
fir	Finite impulse response filter (signal processing algorithms) over a 700 items long sample.	276	4	2
insertsort	Insertion sort on a reversed array of size 10.	92	3	2
janne_complex	Nested loop program.	64	4	2
jfdctint	Discrete-cosine transformation on 8x8 pixel block.	375	5	2
lcdnum	Read ten values, output half to LCD.	64	3	1
ludcmp	LU decomposition algorithm.	147	14	11
matmult	Matrix multiplication of two 20x20 matrices.	163	12	7
ndes	Complex embedded code. A lot of bit manipulation, shifts, array and matrix calculations.	231	25	12
ns	Search in a multi-dimensional array.	535	6	4
nsichneu	Simulate an extended Petri net. Automatically generated code with more than 250 if-statements.	4253	2	2
qsort-exam	Linear equations by LU decomposition.	121	8	6
qurt	Root computation of quadratic equations.	166	16	3
select	A function to select the Nth largest number in a floating point array.	114	6	4
statemate	Automatically generated code.	1276	9	1

Table 1. Benchmark programs used

We have used programs from the Mälardalen WCET Benchmark to test our calculations. Table 1 gives some basic data about the programs (LC = lines of code), number of iteration scopes (#S), and number of (context-dependant) loops (#L). Table 2 shows the results of the different analyses. It shows the following information: Analysis time in seconds for abstract execution with loop bound analysis only (LB), number of found flow facts (#FF), and analysis time (Time) for each of the three algorithms (IN = infeasible nodes, EP = exclusive pairs, IP = infeasible paths). All measurements were performed on a 1.25 MHz PowerPC G4 processor, 1 Gb memory running Mac OS 10.4.6.

We see that we, with a small extra cost, can find infeasible nodes and paths for some of the benchmarks. It

Program	Time	Alg. 1 (IN)		Alg. 2 (EP)		Alg. 3 (IP)	
	LB	#FF	Time	#FF	Time	#FF	Time
adpcm	19.14	37	19.86	44	19.22	24	20.04
bs	0.02	0	0.02	0	0.01	0	0.01
bsort100	0.95	3	0.95	0	0.95	0	0.96
cnt	0.21	1	0.22	0	0.21	0	0.23
compress	0.58	63	0.61	9	0.59	6	0.58
cover	0.71	114	1.65	1061	0.85	102	0.87
crc	2.13	18	2.36	6	2.16	4	2.24
duff	0.05	41	0.06	0	0.06	0	0.06
edn	1.22	0	1.23	0	1.23	0	1.29
expint	0.08	5	0.08	0	0.09	1	0.09
fdct	0.01	14	0.01	0	0.01	0	0.01
fft1	0.19	102	0.23	2	0.19	2	0.19
fibcall	0.02	0	0.02	0	0.02	0	0.02
fir	0.22	1	0.22	1	0.21	1	0.22
insertsort	0.13	0	0.13	0	0.13	0	0.12
janne_complex	0.02	1	0.03	4	0.03	0	0.02
jfdctint	0.03	0	0.03	0	0.03	0	0.03
lcdnum	0.01	41	0.02	6	0.02	6	0.02
ludcmp	1.88	3	1.88	1	1.89	1	1.88
matmult	2.76	0	2.79	0	2.84	0	2.99
ndes	8.02	11	9.39	3	8.10	1	8.13
ns	1.00	1	1.01	0	1.01	0	1.05
nsichneu	12.88	126	13.16	78150	1288.76	623	19.15
qsort-exam	0.18	1	0.19	11	0.18	6	0.19
qurt	0.08	27	0.11	7	0.08	5	0.08
select	0.21	2	0.20	8	0.19	14	0.19
statemate	0.14	256	0.15	5	0.13	32	0.13

Table 2. Analysis results

should be noted that these results are based on single-path analysis, i.e., using a single input that leads to a single execution path. We expect more infeasible nodes and paths to be found when we analyse the programs with inputs that leads to multi-path analyses.

7 Conclusions and Future Work

We do think that our results are promising, but they are still somewhat preliminary: the benchmarks used so far are limited to single-path programs, and we only count the number of generated flow facts for infeasible paths. The next step is to extend the evaluation to a larger set of benchmarks, using multi-path analysis, and to also investigate the effect of the derived infeasible path information on the WCET estimate. In particular, we want to try out the algorithms on industrial real-time codes.

We also want to investigate tradeoffs between analysis time and WCET estimate precision. One possibility would be to generate flow information for individual iterations of a scope. This could give tighter WCET estimates, at the expense of longer analysis times. Another possibility is to generate non-context-sensitive flow facts, valid for all different call-sites of a particular function or loop. This will, in general, give less precise WCET estimates, but for a lower analysis cost.

References

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

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

[3] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation; Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, pages 85–108. Springer-Verlag, 2002.

[4] Ting Chen, Tulika Mitra, Abhik Roychoudhury, and Vivy Suhendra. Exploiting branch constraints without exhaustive path enumeration. In Reinhard Wilhelm, editor, *Proc. 5th International Workshop on Worst-Case Execution Time Analysis, (WCET'2005)*, pages 40–43, Palma de Mallorca, July 2005.

[5] Patrick Cousot and Radhia 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*, pages 238–252, Los Angeles, January 1977.

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

[7] O. Eriksson. Evaluation of Static Time Analysis for CC Systems. Master's thesis, Mälardalen University, August 2005.

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

[9] Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Experiences from industrial WCET analysis case studies. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis, (WCET'2005)*, pages 19–22, July 2005.

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

[11] Jan Gustafsson, Andreas Ermedahl, and Björn 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)*, February 2005.

[12] 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)*, June 1999.

[13] Apostolos A. Kountouris. Safe and efficient elimination of infeasible execution paths in WCET estimation. In *Proc. 3rd International Conference on Real-Time Computing Systems and Applications (RTCSA'96)*. IEEE, IEEE Computer Society Press, 1996.

[14] Y. A. Liu and G. Gomez. Automatic accurate time-bound analysis for high-level languages. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, pages 31–40, June 1998.

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

[16] Mälardalen University WCET project homepage, 2006. `www.mrtc.mdh.se/projects/wcet`.

[17] Daniel Sehlberg. Static WCET analysis of task-oriented code for construction vehicles. Master's thesis, Mälardalen University, Västerås, Sweden, October 2005.