A Tool for Automatic Flow Analysis of C-programs for WCET Calculation

Jan Gustafsson, Björn Lisper, Christer Sandberg, and Nerina Bermudo†
Department of Computer Science and Engineering,
Mälardalen University, Västerås, Sweden.
E-mail: {jan.gustafsson, bjorn.lisper, christer.sandberg}@mdh.se,
†nerina@complang.tuwien.ac.at

Abstract

Bounding the Worst Case Execution Time (WCET) of programs is essential for real-time systems. To be able to do WCET calculations, the iteration bounds for loops and recursion must be known. We describe a newly developed prototype tool that calculates these bounds automatically, thereby avoiding the need for manual annotations by the programmer.

The analysis is based on an intermediate code representation, which means that compiler optimized code is analyzed. The choice of intermediate code also allows the analysis to support a number of programming languages. Right now, C programs are targeted.

We also show an example of a program analysis using out method.

Finally we describe future research directions.

1 Introduction

Real-time systems are systems where failures in the time domain are as harmful as in the functional domain. For so called *hard* real-time systems, no deadline violations are allowed, since this is considered as a failure of the system. Therefore, the *Worst Case Execution Time* (WCET) of programs must be known in hard real-time systems. Once the WCET is known, the software execution can be scheduled.

There are basically two methods to find the WCET of a program, *measurement* or *calculation based on static analysis*. Measurements are troublesome, since it is (for programs with some complexity, and where the execution time is input-dependent) often impossible to safely identify the worst-case input data, or to run the program with all inputs. Sometimes, the WCET is estimated using execution time measurements for *some* input data. As has often been pointed out in the WCET literature, the measured WCET estimate obtained in this way may not be safe, i.e., not worst-case at all. Also, a measurement requires a set-up of the current hardware configuration and the result is limited to this. After any change, the measurements have to be redone. For these reasons, calculation methods are preferred over measurements.

A calculated WCET estimate must be safe (i.e., no underestimation is allowed) and tight (i.e., the overestimation must be as small as possible).

1.1. Manual Annotations

A central issue in WCET calculations is to find an upper limit for *iteration bounds* for loops and recursion. If there is no such limit, the execution time (and thus WCET) is not bounded.

If there are infeasible paths in the program (i.e., paths that cannot be taken for any input), information about these may contribute to an even tighter WCET bound.

In most existing WCET calculation methods, these loops and recursion bounds, and information about infeasible paths, are expected to be given as *manual annotations* by the programmer. However, this work is often time-consuming, complex and error-prone.

Our newly developed flow analysis prototype tool uses static analysis to calculate flow information automatically. In this way, we avoid the need for the programmer to insert manual annotations for iteration bounds in the program.

1.2. WCET Tools

There is a need for a WCET tool useful in an industrial setting. A WCET tool would be a part of the real-time programmer's tool box, and could be useful not only for WCET calculations, but also to find bottlenecks, to assess hardware needs, to evaluate algorithms, and more.

In spite of the usefulness of WCET tools, there are only a few tools available on the market. As a consequence of this, measurements are still industrial state-of-the-practice.

A common architecture of a WCET tool is to subdivide it into *flow analysis*, *low-level analysis* and *calculation*.

- The flow analysis calculates the possible flows in the program. The analysis is based on the control flow structure inside the program, and produces information on iteration bounds and infeasible paths.
- The low-level analysis determines the execution time for each instruction in the program. The analysis must take into account effects from hardware features like pipelines and caches.
- The calculation combines the results from the above two analyses into a final calculation step, which produces the WCET of the program.

The flow analysis tool described in this paper is a part of a planned, complete WCET tool [5] with an architecture as above.

The rest of the paper is organized as follows. Section 2 presents related work, and Section 3 presents our prototype tool. Section 4 contains an example, and Section 5 concludes the paper by discussing ideas for future work.

2. Related Work

Healy et al. are able to calculate the number of iterations for certain types of loops in [9]. However, their work is limited to three different types of loops in C.

Healy et al. also proposed a second method in [10]. The ideas are further developed in Healy's thesis [8]. The method is based on a technique to detect value-dependent constraints. These constraints can be used to limit the number of iterations of loops and to find infeasible paths in loops. The constraints are found without any use of manual annotations. There are, however, cases where these constraints cannot be detected and the method fails.

The Bound-T WCET tool [11] from SSF in Finland uses Presburger Arithmetic to define the properties of simple loops. The Omega calculator [13], which implements Presburger Arithmetic, is then used to calculate the number of iterations of the loop.

Syntactical analysis has been mentioned before, but to our knowledge is has not been used in WCET calculation. A similar idea, the use of templates, is mentioned by Patterson on page 11 in [12], where he describes his work on value range propagation.

The most important difference to these related efforts is that our method is general, i.e., it works for *all* constructs in the supported language.

3. The Flow Analysis Prototype Tool

The *input* to our analysis is intermediate code, originating from a C program. The *output* is a set of "*flow facts*" which describe and constrain the possible flows in the program. We use the flow fact format as described in [4]. The flow facts are connected to a *scope graph* of the program, which is a description of the structures of the program, where our flow facts are valid.

The analysis is based on an *intermediate code* representation of the program. The tool is capable of handling code that is optimized by the compiler, which means that we identify the possible flows in the *real, executed code*. This solves a problem faced by many other WCET methods, by bridging the gap between the source code level (where high-level flow analysis often is done), and the low-level code. These two levels can be quite different if heavy optimizations have been applied. The choice of intermediate code also allows the analysis to support other programming languages, including object-oriented languages.

To cope with the control structures generated from C, our analysis is capable of handling unstructured code and recursion.

The analysis is *optimized* using some speed-up techniques. For example, the tool removes all variable occurrences that do not influence control flow directly or indirectly. In this way, the program to analyze will be of reduced size.

Also, the tool syntactically identifies certain loop constructs that can be analyzed using recurrence equations, which gives iteration bounds directly as solutions to the equations. The loops analyzed in this way can be replaced by simpler constructs, thus reducing program size and complexity of the analysis even further.

For the remaining program, containing the more complex loops, the tool uses *abstract interpretation* [2] to calculate iteration bounds. Also this step uses a number of speed-up techniques, for example merging of intermediate results at strategic points during analysis, to avoid state explosion.

3.1. Overview of the Tool

Basically, the analysis of a C program is performed using the steps described in Figure 1.

- Parser[1]. The C code is parsed to produce a New Intermediate Code (NIC) file. The NIC format is an intermediate format developed by the WPO (Whole Program Optimization) project at Uppsala University [14].
- Optimization[2]. The NIC code is optimized.
- NIC code parser. The optimized NIC code is parsed to produce an internal representation. This internal format is the basis for all subsequent analysis steps.

[1] This step is developed outside our project.
[2] This step is developed outside our project.

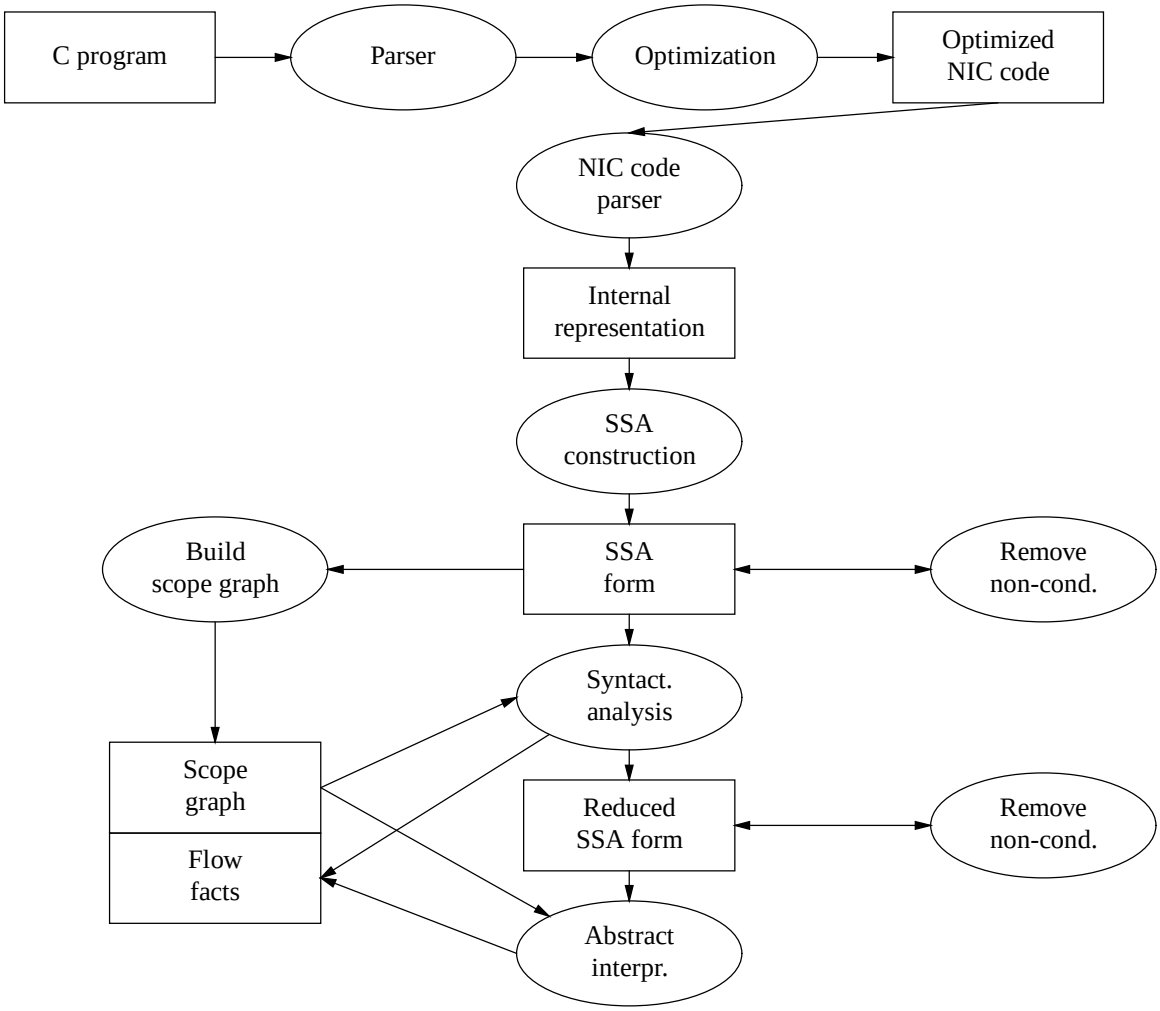

Figure 1. Basic analysis steps.

- An SSA (Static Single Assignment) is constructed for the code (Section 3.2).
- Non-conditionals are removed (Section 3.3). All assignments to variables that do not affect conditions (transitively) are identified and removed from the program.
- Scope graph construction (Section 3.4). The scope graph is constructed using the control flow that can be extracted from the internal representation.
- Syntactical analysis (Section 3.5). The code is "scanned" for a number of simple, recognizable loop constructs and the corresponding loop bounds are calculated, if possible. The resulting flow facts are stored as a result. The analyzed loops are replaced with assignments of the final values to the variables updated in the loop, resulting in a simpler program to analyze in the next step.
- Abstract interpretation (Section 3.6). The remaining code (after the previous step) is analysed using abstract interpretation. The resulting flow facts are appended to the results file.
- If there are constructs for which the abstract interpretation fails, the user is asked for manual annotations for these. The analysis continues with these two last steps until the complete code is successfully analysed.

3.2. SSA Construction

The purpose of this step is to build information about the data flow and the control flow in the program that makes it efficient to analyze and transform the program.

Static Single Assignment (SSA) form [3] is a form of data flow description that is efficient to compute and store (linear in the size of the program). The SSA form encodes the use-def chains (see [1], Section 10.6) which define the dependencies of variables in the program. This information is easily updated when program transformations have been performed.

A program stored in the SSA form has two main properties:

1. Every use of a variable in the program has exactly one reaching definition; and

2. At join points in the program, merge functions called ϕ-functions are introduced.

The SSA format simplifies the subsequent analysis, for example to identify variables that do not influence the control flow, and to identify induction variables in loops.

3.3. Removal of Non-Conditionals

A program contains a number of variables used in, for example, expressions and conditions. Since we are only interested in the control flow of the program (as expressed in the flow facts), we would like to skip all definitions (lvalue) and uses (rvalue) of the variables that do not influence the control flow. In this way, the subsequent analyses and transformations of the program will be more efficient.

The program fragment in Figure 2 is used to illustrate the idea.

```
int i, j, k, n, c, p;
⋮
for (i = 0; i =< n; i = i+c) {
  if (p) then
    k = i + 2;
  }
j = i + k;
c++;
```

Figure 2. Program fragment example

In this program fragment, the only variables that influence the control flow are `i, c, n`, and `p`. Therefore, we can safely remove `k`, `j`, and all their assignments, yielding the reduced program in Figure 3.

Note that it is *variable occurrences* that are analyzed. Therefore, also the incrementing of `c` is removed by our analysis, since the updated value of `c` does not influence the control flow.

```
int i, n, c, p;
⋮
for (i = 0; i =< n; i = i+c) {
  if (p) then {}
  }
```

Figure 3. Program fragment example (reduced)

3.4. Scope Graph Construction

The construction of the scope graph is necessary to be able to define the flow facts of the program. The scope graph concept is used here as described in [4].

A scope graph is created from the control flow graph (CFG) of a program, and extends the CFG to express some of the dynamic behaviour of the program. The scope graph format supports unstructured code and recursion and allows us to store results of various types of flow analysis methods.

Intuitively, a scope corresponds to a certain repeating or differentiating execution environment in the program, such as a function or a loop. A scope can take one of the following forms:

- A set of recursive functions
- A function
- A part of a function:
  - A loop
  - A flow graph with unstructured control flow

As an example, the non-recursive program in Figure 4 yields the complete scope graph in Figure 5. As you can see from the scope graph, the intended analysis is context sensitive, i.e., we differ between different calls to `foo`.

3.5. Syntactical Analysis

Syntactical analysis is based on the observation that many loop constructs have a simple form, for example, they iterate a fixed number of times and do some simple work every iteration. The iteration counts for these loops may be given by *recurrence equations*. If we have closed-form expressions for the solutions of these equations, we can get the iteration counts directly without further analysis of the loop. Assuming we also can find closed-form expressions for the variables updated in the loop, these loops can be replaced by constructs without loops.

The general method that will be used in the tool (*abstract interpretation*) can be time-consuming for loops iterating

```
int main(void) {
int x, y = 0;

for (i = 0, i < 10, i++) {
  y = y + y;
  x = foo(y);
  }
x = foo(x);
return 0;
}

int foo (int i) {
int j, res = 0;

for (j = 0, j < 100, j++)
  if (j > 10) {
  res++;
  }
return(res);
}
```

Figure 4. Code for scope graph example

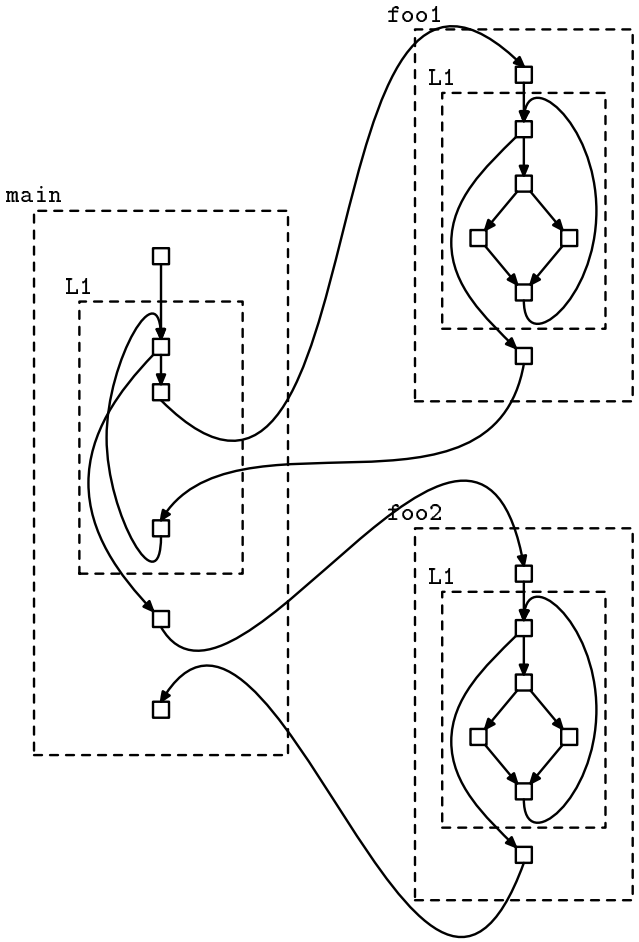

Figure 5. Complete scope graph for the program in Figure 4

many times. Therefore, syntactical analysis and the possible replacement may speed up the flow analysis considerably. How much is, of course, depending on the analyzed program.

Loops in C can take many forms. As a very simple example, regard the loop below.

```
n = 0;
while (n<10) {
  n = n + 2;
  }
```

We can express the semantics of the loop (named L1) using n_i (the value of `n` after iteration i) as the recurrence equation system

$$\begin{cases} n_0 = 0 \\ n_{i+1} = n_i + 2 \quad \text{for } 0 \leq i < x \end{cases}$$

where x is the maximum number of iterations (possibly infinite). This is an "open form" since n_i is used in the right hand side of the equal sign.

The condition $n < 10$ is rewritten to $n_i < 10$ for $0 \leq i < x$. This condition is true within the loop and will be used to calculate x.

There is a closed-form solution to this equation system:

$$n_i = 2i \text{ for } 0 \leq i < x$$

The condition $n_i < 10$ for $0 \leq i < x$ is true within the loop and when it turns to false, the loop terminates. Therefore we are able to calculate x as the smallest x satisfying $n_x \geq 10$ i.e., $x = 5$. This means that we can generate the flow fact

$$\texttt{L1} : [\,] : \#\texttt{header}(\texttt{L1}) = 5$$

for the loop. The meaning of the flow fact is "for the loop L1 the total count for the header is 5" or, in other words, "the iteration bound of loop L1 is 5". Using this fact, we can calculate the value for `n` at loop exit as $n_5 = 2 * 5$ and replace the loop with `n = 10;` thus simplifying the subsequent analysis.

Our analysis will recognize more and more complex loop constructs as the syntactical analysis is developed. This is valid both for calculation of loop bounds and of the final values of index variables (counters) and induction variables.

Loops for which there are no closed solutions, or for which we haven't (yet) implemented the solution, will be passed on to the general abstract interpretation analysis.

3.6. Abstract interpretation

Our aim is to calculate the run-time behavior of a program without having to run it on *all* input data, and while guaranteeing termination of the analysis.

One such technique for program analysis is *abstract interpretation* [2], which means to calculate the program behavior using value descriptions or *abstract values* instead of real values. 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.

The analysis is based on the method as described in, e.g., [6, 7]. The following steps are performed in our tool:

1. The original program is instrumented with *execution count variables*, i.e., variables that keep track of the number of executions of each selected entity (nodes or edges). The values of these counters are set to zero initially and are incremented each time the corresponding entity is visited within the context.
2. An abstract version of the program is analysed using an abstract domain. Currently, we use intervals to represent variable values, but other domains are thinkable.
3. The result is a set of possible values for the execution count variables when the analysis has terminated. This result is expressed using flow facts.

As an illustration, we analyze the program fragment with unstructured control flow shown in Figure 6. It yields the scope graph shown in Figure 7. The inner scope `L1` is an unstructured loop, with both node 2 and 3 as header nodes. Our abstract interpretation will give a safe upper bound on the execution count for these nodes.

Assuming that the possible initial value of a $\in [1..10]$ we will get the following result of the abstract interpretation, expressed as flow facts:

1. $\texttt{main} : [\,] : \#\texttt{header}(\texttt{main}) = 1$
2. $\texttt{main\_L1} : [\,] : \#2 \geq 0$
3. $\texttt{main\_L1} : [\,] : \#2 \leq 5$
4. $\texttt{main\_L1} : [\,] : \#3 \geq 0$
5. $\texttt{main\_L1} : [\,] : \#3 \leq 5$

The meaning of these flow facts is:

```
    if (a < 6) goto l1;
    goto l2;
l1: a := a - 1;
    if (a < 1) goto l3;
    goto l2;
l2: a := (a - 1);
    if (a < 1) goto l3;
    goto l1;
l3: exit
```

Figure 6. Example program with unstructured control flow

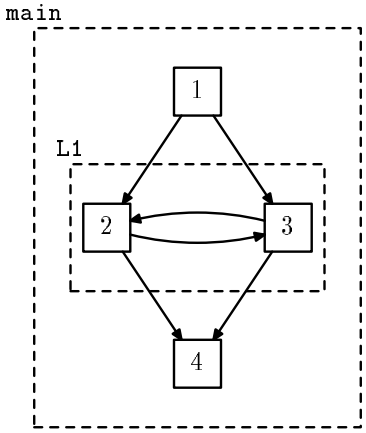

Figure 7. Scope graph of the program fragment in Figure 6

- 1. The whole code is iterated once, i.e., it is not repeated.
- 2, 3. Each time `L1` is entered from `main`, node 2 is executed zero to five times.
- 4, 5. Ditto for node 3.

4. Example

The small C program in Figure 8 has been analyzed by our tool. The result is a scope graph (not shown here) and the flow facts shown below. The first instance of call of `foo` due to the the first call to `foo` is named `foo1` and the second instance is named `foo2`.

1. $\texttt{main} : [\,] : \#\texttt{header}(\texttt{main}) = 1$
2. $\texttt{main\_L1} : [\,] : \#\texttt{header}(\texttt{main\_L1}) = 4$
3. $\texttt{main\_L1\_foo1} : [\,] : \#\texttt{header}(\texttt{main\_L1\_foo1}) = 1$
4. $\texttt{main\_foo2} : [\,] : \#\texttt{header}(\texttt{main\_foo2}) = 1$

```
int foo (int i) {
  int res = 0;
  if (i < 3)
    res++;
  return(res);
  }

int main(void) {
  int x, j, y = 1;
  for (j = 0; j < 3; j++) {
    y = y + y;
    x = foo(y);
    }
  x = foo(x);
  return 0;
  }
```

Figure 8. Example program

The meaning of these flow facts is:

1. `main` iterates once.
2. The loop `L1` in `main` iterates four times.
3. The function `foo` called from loop `L1` in `main` iterates once.
4. The function `foo` called from `main` iterates once.

5. Future Work

Future work includes the following items:

- We will implement *pointer analysis* in the tool.
- *Manual annotations* are currently only supported in a very simple form. We will study how to support these in user-level code.
- *The syntactic analysis* will be developed further.
- Right now the tool is targeted on loop bound calculation. In later versions, identification of *infeasible paths* will be added. The method to discover some types of these are described in, e.g., [7].

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986. Generally known as the "Dragon Book".

[2] P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS'00)*, Nov. 2000.

[5] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 2001.

[6] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. 3rd International European Conference on Parallel Processing, (Euro-Par'97), LNCS 1300*, pages 1298–1307, Aug. 1997.

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

[8] C. Healy. *Automatic Utilization of Constraints for Timing Analysis*. PhD thesis, Florida State University, Florida, USA, August 1999.

[9] 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.

[10] C. Healy and D. Whalley. Tighter timing prediction by automatic detection and exploitation of value-dependent constraints. In *Proceedings of the Fifth IEEE Real-Time Applications Symposium (RTAS'99)*, 1999.

[11] 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)*, Sept. 2000.

[12] J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *SIGPLAN'95 Conference on of Programming Language Design and Implementation (PLDI'95)*, pages 67–78, 1995.

[13] W. Pugh. Counting solutions to presburger formulas: How and why. In *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 121–134, Orlando, FL, June 1994. ACM.

[14] WPO-Whole-Program Optimization WWW Homepage. URL: `http://www.astec.uu.se/etapp3/`, Nov. 2001.