Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis[†]

Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper
Department of Computer Science and Electronics, Mälardalen University
Box 883, S-721 23 Västerås, Sweden
{andreas.ermedahl,christer.sandberg,jan.gustafsson,stefan.bygde,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 for static derivation of precise WCET estimates is upper bounds on the number of times different loops can be iterated.

In this paper we present an approach for deriving upper loop bounds based on a combination of standard program analysis techniques. The idea is to bound the number of different states in the loop which can influence the exit conditions. Given that the loop terminates, this number provides an upper loop bound.

An algorithm based on the approach has been implemented in our WCET analysis tool SWEET. We evaluate the algorithm on a number of standard WCET benchmarks, giving evidence that it is capable to derive valid bounds for many types of loops.

1 Introduction

The WCET is an important parameter when verifying real-time properties. A *static WCET analysis* finds an upper bound to the WCET of a program by analysing the statical properties of the hardware and software involved. Given that the methods used are correct and safe, the analysis will derive a timing estimate that is *safe*, i.e., a value $\geq$ 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 bind 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.

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. Recent industrial WCET case studies [9] have shown that it is important to develop good support for flow analysis, in particular loop bound analysis, in order to reduce the need for manual annotations.

This article presents an approach how to calculate upper loop bounds statically. The approach builds on the observation that terminating loops always must reach a new state for each new iteration. Thus, if we can somehow bound the number of states which are possible to reach during any execution of the loop, then that number provides an upper bound to the number of loop iterations *provided that the loop terminates*. Since in general many states may be equivalent w.r.t. program flow, it suffices to count the number of equivalence classes of states. An upper bound to the number of possible equivalence classes is the number of possible combinations of values for variables affecting the exit conditions of the loop.

Based on this observation, we perform a loop bound analysis using a combination of standard program analysis techniques:

1. By *program slicing* we derive a set of variables and statements that must be considered when deriving a loop bound for a given loop. Only this code is analysed.
2. By *abstract interpretation* (AI) we derive, for each program point and variable, an upper approximation of the possible set of values held by the variable in that program point. This information can be used to limit the possible number of (equivalence classes of) states in loops.

[†] This work has been supported by the KK-foundation through grant 2005/0271. It has also been funded in part by the ARTIST2 Network of Excellence (www.artist-embedded.org).

3. A variable can sometimes have many possible values in a loop, altough its value will always remain the same for each execution of the loop. By *invariant analysis* we identify variables not having their values changed in loops. These variables are removed from the corresponding loop bound calculations.

All the analyses above terminate. Thus, the suggested loop bound analysis also terminates. Moreover, since AI is input dependent, (bounds on input values can be specified), the analysis is also input dependent.

Since our approach assumes that analysed loops terminate, separate proofs of termination will have to be provided. In Section 7 we discuss this problem.

The reminder of this article is organized as follows: Section 2 presents related work. Section 3 presents an illustrating example. Section 4 gives more details of the approach, including some implementation details. Section 5 presents our WCET tool. Section 6 presents some evaluations of the approach. Finally, Section 7 gives our conclusions and ideas for future work.

2 Related work

Upper bounds on the number of loop iterations are needed in order to derive a finite WCET estimate at all. Similarly, recursion depth must also be bounded. Due to the halting problem, no automatic method for loop bounds analysis can give an exact answer for all loops. Thus, WCET analysis tools provide means to give loop iteration bounds manually [5, 6, 17]. However, this is often laborious, and a source of possible errors.

Although necessarily incomplete, an automatic loop bounds analysis can still be useful to reduce the manual work by bounding most of the commonly occurring loops. A common approach is to identify loop counters, and then determine (or bound) their start values, increment (decrement), and highest (or lowest) possible value. From this information, an upper bound for the iteration count can be obtained. Whalley et al. [12] use data flow analysis and specialized algorithms to calculate loop bounds for both single and some special types of nested, triangular loops. This approach is quite syntactical and will fail for loops which do not fit the patterns. The loop-bound analysis of the Bound-T tool [17] estimates range and increment for loop counters using Presburger arithmetics, and the latest loop bound analysis of the aiT tool [4] decides start values by an interval-based AI and the possible increments by a data flow analysis. These methods have in common that they only work for well-structured loops with a proper nesting, and where loop counters are updated using addition or subtraction only. In contrast, our analysis is based entirely on abstract interpretation, which makes is less sensitive to the kind of operations applied to loop counters. It works also for unstructured loops without proper nesting.

```
1.  int foo(int INPUT) {    // INPUT = [10..20]
2.      int OUTPUT = 0;
3.      int i = 1;
4.      while(i <= INPUT) {   // p
5.          OUTPUT += 2;
6.          i++;
7.      }
8.      return OUTPUT;
9.  }
```

Figure 1. Illustrative code example

We have previously presented *Abstract Execution* (AE), a form of symbolic execution based on an AI framework. AE is able to derive loop bounds and infeasible path information for many type of programs [10, 11] but has a potential bad worst-case complexity, and no guaranteed termination. Thus, the approach presented in this article complements the AE by being less general, since it cannot bound all type of loops, but with guaranteed termination.

3 An illustrative example

As an illustrative example, consider `foo` in Figure 1. The `INPUT` variable is given the value limit [10..20].

A slicing w.r.t. the exit condition of the loop discovers that `OUTPUT` does not affect the outcome of the condition, and could, together with statements `2`, `5` and `8`, be sliced away. `i` and `INPUT` are both used in the loop exit condition and must therefore be kept.

An AI using the interval domain together with widening and narrowing (see Section 4.2) will find that at program point `p`, the possible values of `i` lie in the range [1..20], and those of `INPUT` in [10..20]. Thus, a safe bound on the number of times the loop body can be executed, given that the loop terminates, is given by $\texttt{size(i,p)} * \texttt{size(INPUT,p)} = \texttt{size}([1..20]) * \texttt{size}([10..20]) = (20-1+1)*(20-10+1) = 20*11 = 220$. Here $\texttt{size}(v,p)$ is the number of possible values of variable v at a program point p as given by the AI.

We improve the loop bound by observing that the `INPUT` variable is invariant in the loop, although it might assume any value in the range [10..20] in the loop. Therefore, it does not contribute to the calculated loop bound. Thus, a safe upper loop bound, given that the loop terminates, is given by the number of values of `i` at point `p`: $\texttt{size(i,p)} = 20$.

Note that the derived loop bound is input dependent, i.e., another value limitation of the `INPUT` variable could result in a different loop bound.

4 Method details

This section will present our loop bound analysis and its included analyses in more detail. We also present some implementation details, useful for obtaining a faster and more precise analysis.

4.1 Program slicing

Our approach for loop bound analysis uses *program slicing* [18]. Program slicing finds a subset of a program containing the program parts which can affect some given part of the program like a specific condition, or a set of conditions.

Our program slicing works by first building a program dependency graph (PDG) which holds the *data flow* and *control* dependencies between the statements in the program [7, 13]. The slice which is computed with respect to some program part is the part of the PDG which is backwards reachable from the program part. For more details on our program slicing, see [16].

We slice w.r.t. to the exit conditions of the loop to be analyzed. Only the computed slice has to be analysed. In order to reduce the size of the program states of subsequent analyses we also remove variables that are not accessed.

Step-wise slicing Our current implementation actually performs a *stepwise slicing*. First, the program is sliced w.r.t. all conditionals in the program. This removes code that can never affect the outcome of any condition. Then the computed slice is sliced w.r.t. the exit conditions of each single loop to be analysed. Compared to slicing the original program for each loop, this two-step approach gives much better performance, especially if the first slicing is able to remove many statements and variables.

4.2 Abstract interpretation

Abstract interpretation (AI) is a theory of sound approximation of the semantics of computer programs. It was formalized by Cousot & Cousot [2].

AI gives a safe, but potentially pessimistic, estimation of the possible sets of states in different program points. To achieve this, abstract domains with elements representing sets of states, so-called "abstract states" are used. The abstract domains are complete lattices, with a top and a bottom value. For each statement in the language, a corresponding *transfer function* is derived which maps abstract states to abstract states. The transfer functions are used to set up a set of equations relating the abstract states for the different program points. An initial abstract state specifies possible constraints on the input variables. The set of equations is solved using least fixed-point iteration. The least fixed-point defines an abstract state for each program point, and each abstract state represents a safe overapproximation of the set of states in its program point. Often, the abstract states are mappings from program variables to *abstract values* representing possible sets of "concrete" values held by the variables.

For certain abstract domains, the fixed-point iteration will not always terminate. Termination can however be guaranteed through a binary *widening operator* on abstract states, which will enlarge the abstract states during the iteration [2]. Widening can also be used to speed up termination. The solution obtained using widening will be safe, but maybe not the least one. It can sometimes be improved using a *narrowing operator* [3].

Supported abstract domains Our current implementation supports two abstract domains, namely the *interval-* [10] and the *congruence* domain [1, 8]. It also supports the *product* domain of these two domains.

In the interval domain the possible values of a variable is approximated by an interval $[l..u]$. E.g., an abstract state holding the assignment `i` $= [1..20]$ represents all concrete states where $1 \leq$ `i` ≤ 20, i.e., 20 different states.

In the congruence domain the possible values of a variable is approximated by an abstract value of the form $n(\text{mod } m)$. For example, an abstract state holding the assignment `i` $= 0(\text{mod } 5)$ represent all concrete states where `i` contains the factor 5.

The abstract values in the product domain are pairs $\langle i, c\rangle$ where i is an interval and c a congruence. The pair $\langle i, c\rangle$ represents the intersection of i and c. For instance, $\langle [1..20], 0(\text{mod } 5)\rangle$ represents $[1..20] \cap 0(\text{mod } 5) = \{5, 10, 15, 20\}$.

In our implementation each basic data type in C, such as `char`, `int` and `float` has a corresponding abstract data type. We also have abstract versions of aggregate data structures, such a structs and arrays, as well as pointers. Our abstract domains model fixed-size integers with possible overflow. To guarantee termination of the AI we have implemented widening and narrowing operations for our different abstract domains. For details, see [10].

Figure 2 gives an illustrative example of the benefit of using the product domain. An interval analysis would derive `i` $= [0..9]$ at point `p`, corresponding to 10 concrete values. Similarly, a congruence analysis would derive `i` $= 0(\text{mod } 2)$ at point `p` corresponding to an infinite number of concrete values. However, the intersection of the two domains contains all values between 0 and 9 evenly dividable by 2, i.e., $\{0, 2, 4, 6, 8\}$. This set has the size 5, which is a precise loop bound.

```
1.   int i = 0;
2.   while(i < 10) {
3.       // p
3.       i += 2;
4.   }
```

Interval analysis:
$i = [0..9]$ at p

Congruence analysis:
$i = 0 (\text{mod } 2)$ at p

Figure 2. Interval and congruence example

4.3 Loop bound calculation

We use the result of the AI analysis of the sliced program to derive a loop bound for the selected loop. We first select a program point guaranteed to be within the loop, which all iterations of the loop are guaranteed to pass, e.g., the program point just before the last instruction in the loop header node[1]. Then, for all variables not ruled out by the analyses in Section 4.4, the sizes of their respective abstract values are taken as upper bounds to their numbers of possible concrete values. The loop bound is finally calculated by multiplying all these sizes.

For integer and pointer variables, the size of the set of concrete values defined by an interval, or an element in the product domain, is straightforward to compute. The same holds for aggregate objects (array, struct) containing only fields of integer and pointer type. If the variable is or contains a floating-point value then we consider the number of concrete values to be either zero, one or infinite.

If any variable in the abstract state holds the top value, then the loop bound cannot be derived (we consider top to represent an infinite set of concrete values). Similarly, if some variable in the abstract state holds the bottom value, then the loop body is unreachable and we set the loop bound to zero.

4.4 Invariant analysis

Invariant analysis is a program analysis used in many compilers [15]. It identifies statements in loops which can be moved outside the loop since they always recompute the same value. We have implemented a simplified version, which simply checks if any variable used in the (sliced) loop body is also possibly written in the body. A variable that cannot be written is considered loop invariant and can safely be excluded from the loop bounds calculation. Statements reachable through function calls must also be considered. Since pointers can be used to update values, e.g., in `int* p = &i; *p = 5;` variable `i` is assigned a value through the pointer `p`, we use the result of a pointer analysis to find which variables that could possibly be updated through dereferenced pointers.

[1] Assuming, for simplicity, that the loop is well-structured.

```
int i = 1;
while(i <= 100) {
    j = 1;
    while(j <= i) // p
       j++;
    i++;
}
```

(a) Nested loops

```
int temp;  // no init
int j = 0;
while(j < 100) {
    temp = 1;
    j = j + temp;
    temp = 2;
}
```

(b) Problematic code

Figure 3. Invariant analysis examples

Figure 3(a) gives an example where the invariant analysis helps producing a tighter loop bound. At program point `p` an AI using intervals would derive $i = [1..100]$ and $j = [1..100]$. This gives a loop bound of $100 * 100 = 10000$ of the inner loop. However, `i` is invariant in the inner loop, giving that the loop bound can be calculated using `j`'s abstract value only. This gives a loop bound of 100 for the inner loop.

Single-valued-uses analysis The condition detected by the invariant analysis, that a variable never is assigned a new value in the loop body, is unnecessarily strong. Actually, to remove the variable from the loop bounds calculation it suffices that *in any program point in the loop body where the variable might be used, it can hold at most a single value for a given execution of the loop.* An example is shown in Figure 3(b): here, an invariant analysis will fail since `temp` is reassigned two times in the loop, and yet it will always have the single value 1 when used.

We have implemented an analysis to discover if a variable only can have a single value at each relevant use. The analysis simply uses the abstract value derived by the AI, for each relevant variable and program point within the sliced loop body where it is used, to see if the variable can only hold a single value in that point. If this is true for all these program points the variable is removed from the loop bounds calculation.

5 The SWEET tool

SWEET (SWEdish Execution time Tool) [5, 10] is a research tool developed at Mälardalen University [14]. SWEET can handle ANSI-C programs including pointers, unstructured code, and recursion. The basic analysis steps of SWEET are depicted in Figure 4.

Unlike most WCET analysis tools, SWEET is integrated with a compiler and performs its flow analysis on the intermediate representation (IR) of the compiler. The control structure of the IR and the object code is similar, and flow analysis results for the IR, in terms of execution bounds on basic blocks, is therefore also applicable for the object code. The low-level analysis of SWEET currently supports the NECV850E and

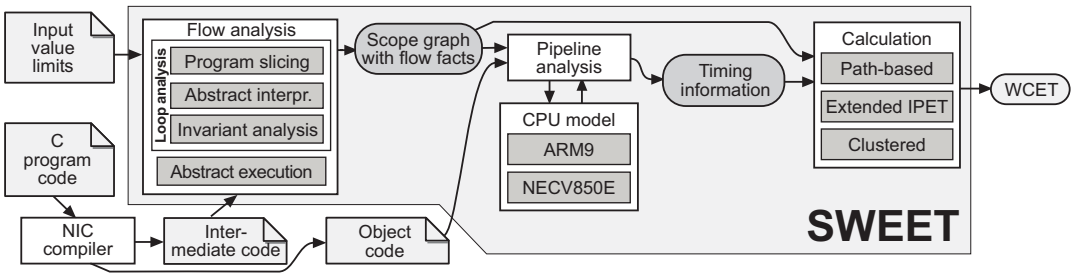

Figure 4. The SWEET WCET analysis tool

ARM9 processors. SWEET supports three different calculation methods: a path-based method, an IPET method, and a hybrid clustered method [5].

The loop bound analysis presented in this article is one of several analyses performed in the flow analysis phase. There is an annotation language which can be used to assign abstract values in the interval domain.

6 Measurements and evaluations

We have used programs from the Mälardalen WCET Benchmark suite [14] to test our flow analyses. The benchmarks are a diverse collection of test programs differing in types of flows, code structure and instructions, intended to thoroughly test different aspects of WCET analysis including flow analysis. Our current implementation of the loop bound analysis cannot handle recursive code, due to limitations in our AI. Thus, no recursive program has been used in our evaluations.

Table 1 gives some basic data about the programs, including lines of C code (**#LC**) and the number of loops (**#L**). The number of loops is counted in a context dependent manner since a loop might have different upper bounds depending from where its corresponding function is called, e.g., `crc` contains such an input dependent loop. For each benchmark we give the number (**#B**) and the percentage (**%B**) of loops bound by the analysis. We also give the number (**#E**) and the percentage (**%E**) of loops which are exactly bound, i.e., given a bound equal to the actual loop bound. The column (**Time**) gives the analysis time in seconds on a 3 GHz PC running Linux.

The **Total** row summarizes our analysis results. We see that more than 60% of all loops gets upper bounded and more than 50% are given an exact loop bound. The loops bounded are in most cases rather simple loops usually dependent on one or two integer index variables. For more complex loops, or loops containing floating point index variables, the analysis often fails.

The analysis time of the loop bound analysis depends very much on how much of the program that could be removed by the slicing. A large remaining program means that the AI usually will take quite a long time. For programs which take long time to analyse, like `adpcm`, `ns`, and `ludcmp`, the analysis time is dominated by the AI.

The results in the table are based on analysis using the interval domain. If we use the product domain described in Section 4.2, we are able to get tighter loop bounds for 6 loops.

7 Conclusions and future work

We have presented a static loop bound analysis based on a combination of standard program analysis techniques. The method has shown to be powerful, giving exact loop bounds for more than 50% of our used benchmarks, with reasonable analysis time.

For future work, we plan to extend the approach to handle more type of loops. One idea is to look into more powerful relational abstract domains in the AI, allowing constraints between values of variables. This should allow the size of the abstract states used for loop bound analysis to be minimized.

We plan to extend the approach to discover if a loop terminates. For example, if it can be shown that each loop variable is either monotonically increasing or decreasing, that no wrap-arounds of these variables could occur, that for any iteration of the loop at least one of the loop variables is updated, and we have a bound on the number of concrete states in the loop, the loop should terminate. We also plan to extend the approach to derive infeasible path information, i.e., paths never possible to execute within a loop body. The latter will be a combination of program slicing, AI and AE.

References

[1] S. Bygde. Abstract interpretation and abstract domains. Master's thesis, Mälardalen University, June 2006.

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

[3] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpre-

Program	Description	#LC	#L	#B	%B	#E	%E	Time
adpcm	Adaptive pulse code modulation algorithm.	879	27	18	67%	8	30%	48.6
bs	Binary search in an array of 15 integer elements.	114	1	0	0%	0	0%	0.81
cnt	Counts non-negative numbers in a matrix.	267	4	4	100%	4	100%	0.24
cover	Program for testing many paths.	640	3	3	100%	3	100%	0.32
crc	Cyclic redundancy check computation on 40 data bytes.	128	6	6	100%	6	100%	0.11
duff	Using "Duff's device" to copy 43 byte array.	86	2	1	50%	1	50%	0.04
edn	Finite Impulse Response (FIR) filter calculations.	285	12	12	100%	9	75%	0.71
expint	Series expansion computing an exponential integral.	157	3	3	100%	3	100%	0.04
fac	Recursive program to calculate factorials.	21	1	1	100%	1	100%	0.01
fdct	Fast Discrete Cosine Transform.	239	2	2	100%	2	100%	0.05
fft1	Fast Fourier Transform using Cooly-Turkey algorithm.	219	30	7	23%	3	10%	5.39
fibcall	Iterative Fibonacci, used to calculate fib(30).	72	1	1	100%	1	100%	0.01
fir	Finite impulse response filter (signal processing).	276	2	2	100%	1	50%	0.38
inssort	Insertion sort on a reversed array of size 10.	92	2	1	50%	1	50%	0.54
jcomplex	Nested loop program.	64	2	0	0%	0	0%	0.04
jfdctint	Discrete-cosine transformation on 8x8 pixel block.	375	3	3	100%	3	100%	0.06
lcdnum	Read ten values, output half to LCD.	64	1	1	100%	1	100%	0.01
ludcmp	LU decomposition algorithm.	147	11	6	55%	5	45%	247.6
matmult	Matrix multiplication of two 20x20 matrices.	163	7	7	100%	7	100%	0.51
ndes	Embedded code with many complex bit operations.	231	12	12	100%	12	100%	3.11
ns	Search in a multi-dimensional array.	535	4	1	25%	1	25%	91.9
nsichneu	Simulates an extended Petri net.	4253	1	1	100%	1	100%	1.11
prime	Search in a multi-dimensional array.	535	2	0	0%	0	0%	0.05
qsort-exam	Linear equations by LU decomposition.	121	6	0	0%	0	0%	76.4
qurt	Root computation of quadratic equations.	166	3	1	33%	1	33%	0.09
select	Selects the n:th largest number in floating point array.	114	4	0	0%	0	0%	19.6
statemate	Automatic generated code.	1276	1	0	0%	0	0%	1.00
ud	Linear equations by LU decomposition.	161	11	11	100%	10	91%	0.53
Total		-	164	104	63%	84	51%	-

Table 1. Benchmark programs and result of loop bound analysis

tation. In *Proc. 4^{th} International Symposium on Programming Languages, Implementations, Logics, and Programs*, Lecture Notes in Computer Science (LNCS) 631, pages 269–295. Springer-Verlag, August 1992.

[4] C. Cullman. Statische berechnung sicherer schleifengrenzen auf maschinencode. Master's thesis, Universität d. Saarlandes, 2006.

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

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

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[8] P. Granger. Static Analysis of Arithmetical Congruences. *International Journal of Computer Mathematics*, pages 165–199, 1989.

[9] J. Gustafsson and A. Ermedahl. Experiences from applying WCET analysis in industrial settings. In *The 10th IEEE International Symposium on Oject/component/service-oriented Real-time distributed Computing (ISORC2007)*, Santorini Island, Greece, May 2007.

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

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

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

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

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

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

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

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

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