Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis using Abstract Execution

Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper
Department of Computer Science and Electronics, Mälardalen University
Box 883, S-721 23 Västerås, Sweden
{jan.gustafsson,andreas.ermedahl,christer.sandberg,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 statically deriving safe and tight WCET bounds is information on the possible program flow through the program. Such flow information can be provided manually by user annotations, or automatically by a flow analysis. To make WCET analysis as simple and safe as possible, it should preferably be automatically derived, with no or very limited user interaction.

In this paper we present a method for deriving such flow information called abstract execution. *This method can automatically calculate loop bounds, bounds for including nested loops, as well as many types of infeasible paths. Our evaluations show that it can calculate WCET estimates automatically, without any user annotations, for a range of benchmark programs, and that our techniques for nested loops and infeasible paths sometimes can give substantially better WCET estimates than using loop bounds analysis only.*

1 Introduction

The *worst-case execution time* (WCET) is an important parameter when verifying real-time properties. A *static WCET analysis* finds an upper bound to the WCET of a program from mathematical models of the hard- and software involved. If 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.

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.

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 possible input data values. In contrast to loop bounds, infeasible path information is not required to find a WCET estimate, but may tighten the resulting WCET estimate.

Recent industrial WCET case studies [3, 7, 20], have shown that it is important to develop good support for both loop bound and infeasible path analyses, thereby reducing the need for manual annotations.

This article presents methods to automatically calculate loop bounds and infeasible paths. The methods have all been implemented in our prototype WCET tool SWEET (SWEdish Execution time Tool). The concrete contributions of this article are:

- We present our analysis method, *abstract execution*, which calculates various kinds of flow information.
- We present several methods to calculate loop bounds and different types of infeasible paths, allowing us to trade analysis time for WCET estimate precision.
- We evaluate the effects of our different methods, in terms of analysis time, generated flow information and impact on WCET estimate for a number of WCET benchmarks.

The rest of this paper is organized as follows: Section 2 discusses different types of loop bounds and infeasible paths, and describes related work. Sections 3 and 4

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

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

present our representation for program flow and the basic flow analysis, while Section 5 describes how the analysis is adapted to derive loop bounds and infeasible paths. Section 6 gives an illustrating example. Section 7 presents SWEET, and Section 8 gives analysis results and evaluations. In Section 9 we draw conclusions and point out directions for further work.

2 Flow Analysis and Related Work

Any WCET analysis must deal with the fact that most computer programs do not have a fixed execution time. *Variations* in the execution time occur due to different input data, and the hard- and software characteristics. Thus, both inputs as well as hard- and software properties must be considered in order to derive a WCET estimate.

Consequently, static WCET analysis is usually divided into three phases: a *flow analysis*, a *low-level analysis* where the execution times for sequences of instructions are decided from a performance model for the hardware, and a final *calculation* phase where the flow and timing information is combined to yield a WCET estimate.

The purpose of the flow analysis is to derive information about the possible execution paths through the program. To find exact flow information is in general undecidable[1]: thus, any flow analysis must be approximate. To ensure a safe WCET estimate, the flow information must include all feasible program paths.

In low-level analysis, researchers have studied effects of various hardware enhancing features, like caches, branch predictors and pipelines [6, 17, 21]. A frequently used calculation method is IPET (Implicit Path Enumeration Technique), using arithmetical constraints to model the program structure, the program flow and low-level execution times [8, 15, 21].

2.1 Loop Bounds

Upper bounds on the number of loop iterations are needed in order to derive a WCET estimate at all. Similarly, recursion depth must also be upper bounded. Since these bounds are not explicitly given in the program code, WCET analysis tools provide ways to give them manually [22, 10, 8]. However, this is often laborious, and a source of possible errors.

Consequently, flow analysis research has mostly focused on automatic *loop bound analysis*. The aiT WCET tool [21] has a loop-bound analysis using a combination of an interval-based abstract interpretation and pattern-matching. The loop-bound analysis of the Bound-T WCET tool [22] is based on Presburger arithmetics. The loop bounds can be derived in a context sensitive manner, which may yield different bounds for the same loop depending on the calling context. Whalley et al. [13] use data flow analysis and specialized algorithms to calculate loop bounds for both single and some special types of nested, triangular loops.

2.2 Infeasible Paths

Infeasible path information is not required to find a WCET bound, but may tighten the estimate. An extreme case of an infeasible path is dead code. Infeasible paths can be caused by semantic dependencies, 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 in conflict[2], and the corresponding path **A-C** can never be taken. This type of infeasible path can be found using some kind of semantic analysis.

Limitations of input data values may yield more infeasible paths. For instance, if we know that `x>5` when the above code fragment is executed, then we can conclude that the paths **A-C**, **A-D**, and **B-D** are all infeasible. This kind of infeasible path can be found by an input-sensitive semantic analysis.

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 [16] studies detection of infeasible paths in the synchronous real-time language SIGNAL. Healy et al. [14] use value-dependent constraints to find infeasible paths. Aljifri et al. [1] generate only the feasible paths using the concept of partially-known variables. Chen et al. [4] finds infeasible paths by identifying conflicts between variable assignments and branch conditions. The symbolic simulation method of Lundqvist [17] also detects some infeasible paths.

The abstract execution, which is the basis for our flow analysis methods, has some similarities with the method of Lundqvist [17]. However, our abstract execution uses a more detailed value domain (see e.g., [12]) and is based on an abstract interpretation framework.

3 Scope Graphs and Flow Facts

The flow analysis presented here uses a *scope-graph* [8]. Each scope is a program part which may be repeated, such as a function or a loop. For each scope, there is a subset of its CFG nodes of which exactly one node must be executed for each *iteration* of

[1] A perfect flow analysis would solve the halting problem.

[2] We assume that `x` is not updated in **A** and **B**.

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

the scope: typically, this subset consists of the header node of a well-structured loop, or the entry node of a function. For each scope, some flow information is collected during the analysis: see Section 5.

A scope graph describes how scopes are invoking other scopes. Our current scope-graph representation is context-sensitive, i.e., a scope is statically created for each call site to a function, or loop. This means that calls to a function at different call sites are analysed separately, which may yield higher precision but also a costlier analysis. The scope graph is acyclic, and expresses a "containment" relation between scopes: for instance, a loop nest will be represented by a chain of scopes where scopes for inner loops are below scopes for outer loops (see Figure 6 for an example).

The purpose of scope graphs is to structure the flow analysis, and the generated flow constraints, such that the execution of repeating constructs can be analyzed and constrained. Flow information for scopes can be expressed as *flow facts* [8]. Flow facts constrain virtual *execution counters* for the CFG nodes in a scope: for each node `B`, its counter `#B` is initialized to zero each time a scope is entered from above, and incremented at each execution of `B`.

The current flow fact language allows linear inequalities on the counters, as well as constructs to restrict constraints to certain scope iterations. Flow facts have the format *scope* : *context* : *linear_constraint*. Here, *context* is either a *forall* context [*range*], specifying that for *all* iterations of *scope*, *linear_constraint* should hold, or a *foreach* context <*range*>, specifying that for *each individual* iteration of *scope*, *linear_constraint* should hold. If *range* is left out, then the constraint should hold for all possible iterations. Loop bound and infeasible path constraints are easily expressed as flow facts: for a loop "`loop`" with header node `H` and nodes `B1`,...,`Bn` the flow fact

```
loop : [ ] : #H < k
```

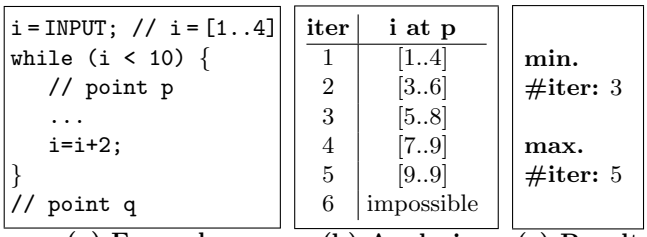
restricts the number of loop iterations to at most $k-1$, whereas the flow fact

```
loop : <3..7> : #B1 + ... + #Bn < n
```

states that for each of the individual loop iterations `3` to `7`, all the nodes `B1`,...,`Bn` cannot be executed during the same iteration.

4 Abstract Execution

Abstract execution is a form of symbolic execution [11, 12], which is based on abstract interpretation. 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: thus, For instance, the abstract domain can



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

Figure 1. Example of abstract execution

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. Figure 1 illustrates how abstract execution works for a loop, by iterating, and counting the number of iterations, until the abstract version of the loop condition surely returns `false` for the back edge.

Abstract execution analyzes all executions in a certain program point separately. This is different from traditional abstract interpretation, where the abstract state for a program point typically covers all concrete states in that program point, for all executions [5].

The abstract interpretation framework guarantees that a calculated abstract value always represents a set including the 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. For the infeasible path analyses this means that some infeasible paths might be reported as feasible. However, this is safe, since a larger set of feasible paths only gives a possibly less tight WCET estimate.

Sometimes, abstract execution of a condition node will yield possible execution paths for both the `true`- and `false`-branch. In Figure 1 this occurs for the test of the loop condition before iteration 4 and 5, where `i = [7..10]` and `[9..11]`, respectively. Two abstract states will then be created, one for each outcome of the test. This means that abstract execution may have to handle many abstract states, representing different possible execution paths, concurrently. The number of possible abstract states may grow exponentially with the length of these paths: thus, any algorithm for abstract execution must be able to *merge* abstract states, which is typically done at program points where different paths join. If the states are merged using the least upper bound operator "$\sqcup$" on the abstract domain of states, then the result is an abstract state safely repre-

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

```
FOREACH sc in scopes DO
  init(sc.c); /* initialize collector */
wlist <- {init_state};
merge_list <- empty;
final_list <- empty;
REPEAT
  WHILE wlist /= empty DO {
    s <- select_from(wlist);
    wlist <- wlist \ {s};
    new_states <- ae(s);
    FOREACH s' in new_states DO
      IF update_point(s')
      THEN sc(s').c <- update(s'.r,sc(s').c);
      CASE merge_point(s'): merge_list <-
                              merge_list U {s'}
           final_state(s'): final_states <-
                              final_states U {s'}
           otherwise: wlist <- wlist U {s'};
  }
  WHILE merge_list /= empty DO {
    s <- select_from(merge_list);
    merge_list <- merge_list \ {s};
    FOREACH s' in merge_list DO
      IF same_merge_point(s,s') THEN
        s <- merge(s,s');
        merge_list <- merge_list \ {s'};
    wlist <- wlist U {s};
  }
UNTIL wlist = empty
```

Figure 2. Algorithm for abstract execution.

senting all possible concrete states, but possibly with some loss of precision. Different strategies for merging will thus yield different tradeoffs beween analysis time and precision.

We have designed and implemented an algorithm for abstract execution, which can generate different kinds of flow facts. The algorithm is described in Figure 2. It is a quite straightforward worklist algorithm, which iterates over a set of abstract states, generating new abstract states from old ones. Abstract states at merge points are moved to a special merge list, and final states are removed. When the worklist is empty, all states in the merge list which are at the same merge point are merged, and the resulting states are inserted in the worklist. The algorithm terminates when both the merge list and the worklist are empty. This algorithm is not guaranteed to terminate for all programs, so in practice it is augmented with a timeout mechanism which gives the user information on the currently analysed loop. This may be useful, e.g., to find loops that overruns the number of iterations.

An abstract state s is a 4-tuple $(s.i, s.r, s.p, s.\sigma)$. $s.i$ is a stack of current *iteration counts* for the current and all surrounding scopes. For each new iteration of the current scope, the top element is incremented. If a scope is exited, then the top element is popped, and if a scope is entered then 1 is pushed onto the stack. $s.r$ is a *recorder* which is used to collect information for generating flow constraints: see Section 5. $s.p$ is the current program point, and $s.\sigma$ is an *abstract store* describing the possible memory contents. Our abstract stores are mappings from addresses to intervals, but other abstract domains can also be used. The function `ae` performs a step of abstract execution, and updates the components of the abstract state accordingly. Note that for abstract execution of a condition, more than one new state may be generated. The abstract execution starts with a single initial state, whose abstract store may specify restrictions on the input values for the analyzed program: thus, the algorithm is input-sensitive.

Each scope sc has a *collector* $sc.c$, which collects information from the abstract states for generating flow constraints, see Section 5. When an abstract state `s` reaches an *update point*, the collector of the scope `sc(s)` for `s` is updated using the recorder of `s`. Typically, update points occur at new iterations, and scope exits. The algorithm is asynchronous since that updates may occur in different order, depending on when abstract states to execute are drawn from the worklist. Also, states at merge points may be "released" before all states to merge have arrived to the point: this is a deliberate design decision to keep the average size of the merge list down, but it means that the algorithm sometimes might return more precise results by avoiding some merges, at the expense of exploring more paths. The result will however always be safe.

The worklist algorithm allows merge between arbitrary states. However, in our algorithms in Section 5, merging takes place only for abstract states at the same scope iteration and program point. For such states, merging is defined as:

$$(i, r, p, \sigma) \sqcup (i, r', p, \sigma') = (i, r \sqcup r', p, \sigma \sqcup \sigma')$$

5 Calculation of Flow Information

We now present our methods to calculate loop bounds and infeasible paths. All the methods are variations of the abstract execution described in Section 4, with different recorders, collectors, update points, and update operations. The abstract state transition function `ae` also differs in its handling of recorders depending on which method is used. The methods can be freely combined. At the end of the analysis, each collector is used to generate flow facts for its scope.

5.1 Loop Bound Calculation

This analysis finds safe lower and upper bounds to the total number of iterations of a scope (typically a loop). The collector is a pair (l, u) which is initialized to $(\infty, 0)$. The recorder is a number, which is a virtual execution counter for the header node(s) of the scope: this counter is set to one at entry to the scope, and is incremented by one for each new iteration. The update points are the exits from the scope, and the collector is updated as

$$\texttt{update}(r,(l,u)) = (\texttt{min}(r,l),\texttt{max}(r,u))$$

A recorder is incremented exactly when the scope iteration count is. Since we only merge states with the same iteration count, we thus only need to perform the trivial merge $r \sqcup r = r$ of recorders.

For the abstract execution example in Figure 1, abstract states with loop header execution counters 3, 4, and 5 will appear at the loop exit point, which yields the final value of $(3, 5)$ for the computed loop bound.

5.2 Loop Bounds for Inner Loops

The analysis in Section 5.1 calculates maximal and minimal iteration counts for a loop. For nested loops like triangular loops, where the iteration count of the inner loop varies with the iteration number of the outer loop, this leads to overestimations for the total number of iterations of the inner loop. For such an inner loop, it is better to accumulate the total sum of the number of iterations in the context of the outer loop.

This can be done by a simple variation of the loop bounds analysis above. Consider an outer loop with scope `l_outer`, containing, at some nesting level, an inner loop with scope `l_inner`. The method works almost exactly as the analysis in Section 5.1 applied to `l_outer`: the only difference is that the recorder is incremented at each iteration of `l_inner`. This will compute minimum and maximum of the total number of iterations of the inner loop, for any single iteration of the outer loop. The analysis in Section 5.1 can be seen as a special case, with `l_inner` = `l_outer`.

5.3 Detecting Infeasible Nodes

Infeasible nodes are never visited in any execution of a certain scope. For this analysis, the recorder is a bit array with one bit per node in the scope. These bits are all set to zero when entering the scope, and the bit of a node is set to one at each abstract execution of the node. Thus, a value of zero means "definitely not executed" and one means "may have been executed".

The collector object is a similar bit array, similarly initialized. The update points are the scope exits. The update operation, as well as merge of recorders, is bitwise `or` of bit arrays. 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 execution of the scope `scope`. An infeasible node is not necessarily the same as dead code, since it is infeasible only w.r.t. a certain scope, and there may be other scopes, for the same code, where the node is executed.

5.4 Upper Bounds for Node Executions

This analysis generalizes the method in Section 5.3. Recorders and collectors are arrays which hold one natural number per node in the scope. The recorder array elements are now execution counters for the nodes: they are all initialized to zero, and incremented at each abstract execution of the node in question. The collector is also initialized to all zeroes, and merge and update is elementwise maximum on arrays.

At termination, if the collector holds a number n for a node, then the node is surely never executed more than n times for each entry of the scope, and a corresponding "upper execution bound flow fact" can be generated. An example is:

```
loop : [ ] : #BB91 <= 7;
```

specifying that basic block `BB91` is never executed more than 7 times in any execution of the scope `loop`.

To avoid unnecessary flow facts, our current implementation generates only such flow facts where the upper bound n is less than the loop bound of the scope. Then, the node has surely been infeasible for some iteration(s).

5.5 Detecting Infeasible Pairs of Nodes

This analysis finds pairs of nodes where both never execute in the same iteration of a scope. Recorders are sets of paths taken through the scope body during an iteration. They are initialized to the set containing the empty list, and nodes are appended to each path in the set when abstractly executed. To limit the size only nodes after conditionals are appended: this still yields unique path representations. Merge of recorders is set union, and the recorders are re-initialized for each new iteration of the scope.

Collectors are triangular $N \times N$-matrices, where N is the number of successor nodes to condition nodes in the scope. The matrix elements can be $\perp$, 0, or 1. Initially, all elements are $\perp$. The update points are immediately before new iterations of the scope, before

```
update(S,M) =
M' := M;
FOREACH path RL in S do
  FOREACH node n1 in RL do
    FOREACH subsequent node n2 to n1 in RL do
      M'[n1,n2] := 1
      FOREACH 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
return M'
```

Figure 3. Infeasible pairs collector update

the recorders are re-initialized, as well as at scope exits. Update of a collector matrix with a recorder set is defined in Figure 3. The assignment using OR gives a safe approximation of the matrix element.

If a position in the final collector holds 0, then the corresponding two nodes can never both be executed in the same iteration, and we can generate an "excluding pair flow fact", like:

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

meaning that the two nodes cannot be executed together in the directly surrounding loop scope `scope`.

The same can be done for positions where the element is $\perp$. However, some of those positions are for pairs of nodes which can never both be executed in an iteration due to the structure of the CFG for the scope body. These flow facts will then be superfluous: the element $\perp$ yields a simple way to avoid their generation without making a reachability analysis in the CFG.

5.6 Detecting Infeasible Paths

This analysis finds sequences of nodes which are never executed together during the same iteration of a scope. The method uses the fact that many infeasible paths can be efficiently represented by allowing them to share a common prefix. As for the infeasible pairs analysis, collectors are updated and recorders re-initialized at new scope iterations and scope exits.

The recorder is now a tree of CFG nodes. Like for the infeasible pairs analysis, we only keep track of nodes taken after branches. However, the tree additionally keeps track of branch outcomes not taken, through a boolean tag for each leaf node. Every path through the tree represents the set of paths for which it is a prefix. For simplicity, we assume that all scope iterations start from the same CFG node (e.g., the header node of a well-structured loop): if not, an artificial root node can be added.

Figure 4 illustrates how the recorder tree works. Figure 4(a) gives a CFG with $2^3 = 8$ structurally possible execution paths. Figure 4(b) shows the tree for

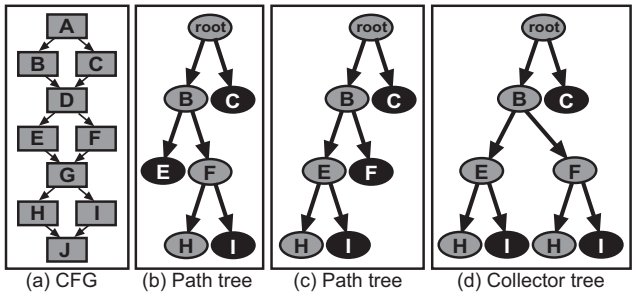


Figure 4. CFG and path tree examples

the path **A-B-D-F-G-H-J** through the CFG. In this tree, the paths **A-C**, **A-B-D-E** and **A-B-D-F-G-I** are marked as infeasible. Similarly, Figure 4(c) shows the tree for the path **A-B-D-E-G-H-J**. Note that the path **A-C** actually represents $2^2 = 4$ paths through the CFG, for which it is a prefix.

Merge of recorders is merge of trees: Figure 4(d) shows the merge of the trees in Figures 4(b) and 4(c). Collectors are the same kind of trees as recorders, and update is the same as merge. We define the set of trees, and merge operation "$\sqcup$" on trees, more formally as follows. A tree is either: the empty tree Λ, a leaf $leaf(n, I)$ or $leaf(n, F)$, where n is a CFG node, I represents an infeasible path and F a feasible path, or an internal node $int(n, t, t')$ where n is a CFG node and t, t' are trees. We then define $\sqcup$ as a commutative operator which also satisfies the following equations:

$$
\begin{aligned}
t \sqcup \Lambda &= t \\
leaf(n, I) \sqcup leaf(n, I) &= leaf(n, I) \\
leaf(n, F) \sqcup leaf(n, x) &= leaf(n, F), \quad x = I, F \\
int(n, t, t') \sqcup leaf(n, x) &= int(n, t, t'), \quad x = I, F \\
int(n, t_1, t'_1) \sqcup int(n, t_2, t'_2) &= int(n, t_1 \sqcup t_2, t'_1 \sqcup t'_2)
\end{aligned}
$$

Note that we can represent all sets of paths by trees built in such a way that we never have to merge trees with different header nodes: thus, we need not define merge of such trees.

Flow facts can be generated for the infeasible paths in the final collector tree, like:

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

specifying that for no iteration of `scope`, all the basic blocks `BB33`, `BB57`, and `BB82` are executed.

6 An Illustrative Example

The example code in Figure 5 contains several types of infeasible paths. The program contains eight scopes; `main`, `foo1`, `foo2` (the two calls to `foo`) and their corresponding loop scopes, `foo1_L` and `foo2_L`, and `bar` and its two nested loops, `bar_L` and `bar_L_L`, as shown in Figure 6. We assume that neither `x` nor `y` is changed in the code.

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

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

Figure 5. Code with several infeasible paths

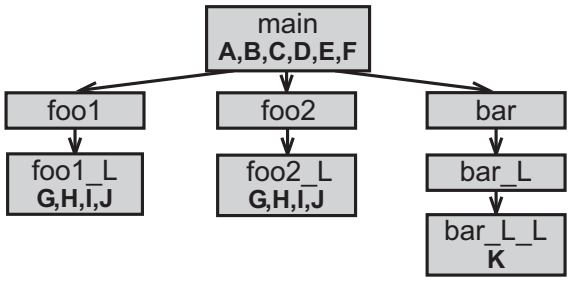


Figure 6. Scopes

Due to mutually exclusive conditions, independently of data and context, we can detect:

1. Infeasible nodes: none.
2. Infeasible pairs: **B-C**, **B-E**, **D-E**, **G-I**, and **H-J**.
3. Infeasible paths: **A-D-E**, **B-C-E**, **B-D-E**, **B-C-F**, **G-I**, and **H-J**.

Due to given input data constraints we also detect:

1. Infeasible nodes: **E**.
2. Infeasible pairs: **A-E**, **C-E**.
3. Infeasible paths: **A-C-E**.

Furthermore, we can identify some context dependent infeasible nodes; **H** and **I** are infeasible in `foo2_L`.

`bar` holds a triangular loop nest. The loop bounds analysis of Section 5.1 gives an upper iteration bound of $100 * 100 = 10000$ for **M**, whereas the analysis in Section 5.2 yields an upper bound of 5050.

7 The SWEET WCET Analysis Tool

SWEET [8, 12] is a WCET analysis research prototype tool developed at Mälardalen University [18]. SWEET can handle full ANSI-C programs including pointers, unstructured code, and recursion. The basic analysis steps of SWEET are shown in Figure 7.

Unlike most WCET analysis tools, SWEET is integrated with a compiler and performs its flow analysis on the intermediate representation (IR) of the compiler, after structural optimizations. Thus, the control structure of the IR and the object code is similar, and the flow analysis for the IR is valid for the object code as well. The low-level analysis of SWEET [6] currently supports the NECV850E and ARM9 processors. The tool currently supports three different calculation methods: a fast path-based method, a global IPET method, and a hybrid clustered method [8, 9]. For the evaluations in Section 8 we used the ARM9 timing model and the IPET calculation method. This timing model has not been validated against real hardware. However, we consider it to be a sufficiently realistic "abstract architecture" for the purpose of evaluating flow analysis methods.

SWEET currently uses abstract execution with intervals, as described in Sections 4 and 5, for the flow analysis. It allows the user explicit control over the placement of merge points, to control the tradeoff between precision and analysis time. The implementation also uses a number of techniques to speed up the analysis. One example is *program slicing*, which is used to restrict the abstract execution to only those program parts that may affect the program flow [19]. It is used in all analyses below to reduce the analysis time.

Program	Description	#LC	#S	#BB
`bs`	Binary search in an array of 15 integer elements.	114	3	34
`cover`	Program for testing many paths.	640	7	1298
`crc`	Cyclic redundancy check computation on 40 bytes of data.	128	11	115
`edn`	Finite Impulse Response (FIR) filter calculations.	285	21	328
`fac`	Recursive program to calculate factorials.	21	4	31
`fdct`	Fast Discrete Cosine Transform.	239	4	143
`fibcall`	Iterative Fibonacci, used to calculate fib(30).	72	3	26
`inssort`	Insertion sort on a reversed array of size 10.	92	3	35
`jcomplex`	Nested loop program.	64	4	41
`lcdnum`	Read ten values, output half to LCD.	64	3	130
`minmax`	Small program with min and max calculations.	32	7	97
`ndes`	Complex embedded code. A lot of bit manipulation, shifts, array and matrix calculations.	231	25	409
`ns`	Search in a multi-dimensional array.	535	6	39
`nsichneu`	Simulates an extended Petri net. Automatically generated code with more than 250 `if`-statements.	4253	2	2686

Table 1. Benchmark programs used

8 Evaluation

We have used programs from the Mälardalen WCET Benchmark suite [18] 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. We only use

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

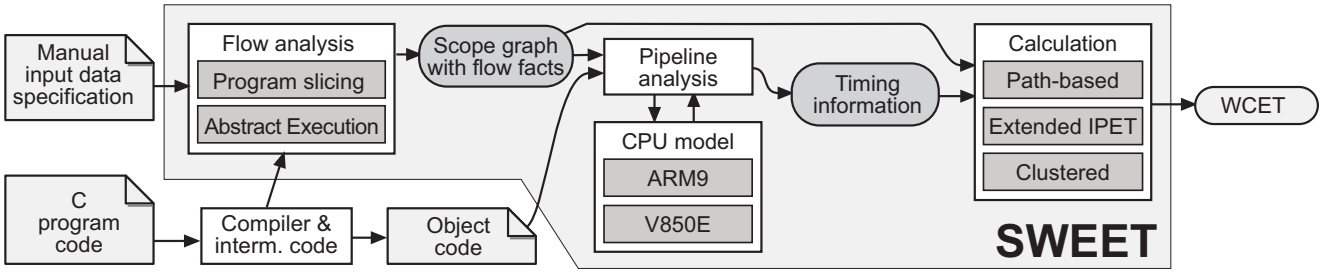


Figure 7. The SWEET WCET analysis tool

Program	#L	#N	LB		LBI			
			Time	WCET	Time	+%	WCET	−%
bs	1	0	0.02	1009	0.02	0	1009	0
cover	3	0	3.65	73128	3.68	1	73128	0
crc	6	0	1.17	834159	1.19	2	834159	0
edn	2	3	0.94	1425085	1.00	6	1425085	0
fac	1	0	0.02	4658	0.02	0	4571	0
fdct	2	0	0.10	40856	0.11	10	40856	0
fibcall	1	0	0.02	3064	0.02	0	3064	0
inssort	2	1	0.07	31163	0.08	14	18167	42
jcomplex	2	1	0.03	12039	0.03	0	2586	79
lcdnum	1	0	0.07	5507	0.09	13	5507	0
minmax	0	0	0.05	563	0.07	17	563	0
ndes	12	0	4.43	795425	4.47	0	795425	0
ns	4	1	0.48	130733	0.48	0	130733	0
nsichneu	2	0	32.37	119707	23.08	-28	119707	0

Table 2. Loop bound analysis results

benchmarks without floating point calculations, since ARM9 lacks hardware support for these. All analyses were done on a PC with a 3 GHz Intel Pentium 4 CPU and 1 GB memory, running Linux 2.6.9-11.

Table 1 gives some basic data about the programs, including lines of C code (#LC), number of scopes (#S) and basic blocks in the ARM9 code (#BB). The analysis times include flow analysis, low-level analysis and WCET calculation. Compilation and parsing times are not included.

Loop bound analysis. Table 2 shows the results of our basic loop bound analysis (LB, see Section 5.1), and our loop bound analysis for nested loops (LBI, see Section 5.2). The table shows the relation between the type of loop analysis, the analysis time in seconds (Time), and the resulting WCET estimate in cycles (WCET). The table also shows the number of (context-dependent) loops (#L) and loop nests (#N) in the program. +% gives the savings in % compared to the WCET for LB. All WCET estimates are calculated for single-path executions of the programs, i.e., using input data corresponding to one execution.

SWEET correctly calculates the maximal number of iterations for all loops in the benchmarks, without using any annotations. This means that our methods can perform a fully automatic WCET analysis for these.

As we can expect, there is no difference between LB and LBI for programs with no nested loops, or when the inner loop bounds are independent of the iteration of the outer as in `edn` and `ns`. In `inssort` and `jcomplex` there are such dependencies, and for these programs the WCET estimate is reduced when LBI is used. This extra tightness is achieved at practically no extra cost in analysis time.

Infeasible path analysis. Table 3 shows the infeasible path analysis results. We have separate tables to show the results for infeasible node analysis (IN, see Section 5.3), upper bounds for node executions (UN, see Section 5.4), exclusive node pairs (EP, see Section 5.5), and infeasible paths (IP, see Section 5.6). The tables show the trade-off between the analysis time for different infeasible path analysis algorithms and the size of the WCET estimate.

The columns show the WCET estimate for LBI (WCET orig., see Table 2), analysis time in seconds (Time), extra time in % compared to the analysis time for LBI (+%), number of flow facts generated (#FF), calculated WCET in ARM 9 clock cycles (WCET), and savings (−%) in % compared to the WCET estimate in column 2. The WCET is calculated for single-path executions of the programs.

The infeasible path analysis yields a reduction in calculated WCET up to around 50% for some programs. The extra analysis time is typically below 50%, with a few exceptions. The EP and IP analyses of `nsichneu` are extreme outliers. The long analysis time is mainly spent in the calculation phase, which obviously is sensitive to the high number and type of flow facts generated. It should be noted, however, that `nsichneu` is a program with extremely many ($2^{250} \approx 10^{75}$) potentially possible paths.

All analyses yield some WCET estimate reduction for some, but not always the same, programs. For some programs, no improvements are achieved. How much improvement, and for which programs, is dependent on the program control logic: programs where some infeasible path analysis yields a large reduction, like `lcdnum` and `nsichneu`, have a control structure with many if-statements and many possible paths.

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

Program	WCET	IN				
	orig.	Time	+%	#FF	WCET	−%
bs	1009	0.02	0	0	1009	0
cover	73128	4.70	28	114	72588	1
crc	834159	1.29	8	18	830278	0
edn	1425085	1.00	0	0	1425085	0
fac	4571	0.02	0	0	4571	0
fdct	40856	0.13	18	0	40856	0
fibcall	3064	0.02	0	0	3064	0
inssort	18167	0.08	0	0	18167	0
jcomplex	2586	0.03	0	1	2586	0
lcdnum	5507	0.09	0	41	4907	11
minmax	563	0.07	0	15	562	0
ndes	795425	4.95	11	11	794145	0
ns	130733	0.48	6	1	130671	0
nsichneu	119707	23.79	1	126	57247	52

Program	WCET	UN				
	orig.	Time	+%	#FF	WCET	−%
bs	1009	0.02	0	8	1002	1
cover	73128	5.57	51	576	63563	13
crc	834159	1.22	3	36	830278	0
edn	1425085	1.04	4	17	1425085	0
fac	4571	0.02	0	30	4292	6
fdct	40856	0.11	17	2	40856	0
fibcall	3064	0.03	0	2	3064	0
inssort	18167	0.08	0	4	18167	0
jcomplex	2586	0.03	0	10	2523	2
lcdnum	5507	0.09	0	53	2902	47
minmax	563	0.07	0	6	563	0
ndes	795425	4.68	5	139	793905	0
ns	130733	0.49	2	7	130671	0
nsichneu	119707	22.54	-4	877	57247	52

Program	WCET	EP				
	orig.	Time	+%	#FF	WCET	−%
bs	1009	0.03	50	0	1009	0
cover	73128	5.29	44	1061	73119	0
crc	834159	1.35	0	6	833301	0
edn	1425085	1.18	0	0	1425085	0
fac	4571	0.02	0	0	4571	0
fdct	40856	0.12	0	0	40856	0
fibcall	3064	0.02	0	0	3064	0
inssort	18167	0.08	0	0	18167	0
jcomplex	2586	0.03	50	4	2586	0
lcdnum	5507	0.11	38	21	5450	1
minmax	563	0.07	0	4	455	19
ndes	795425	4.74	0	3	791980	0
ns	130733	0.51	0	0	130733	0
nsichneu	119707	574.68	2351	78150	119707	0

Program	WCET	IP				
	orig.	Time	+%	#FF	WCET	−%
bs	1009	0.02	0	0	1009	0
cover	73128	4.82	31	102	73128	0
crc	834159	1.24	4	4	833301	0
edn	1425085	1.01	1	0	1425085	0
fac	4571	0.02	0	0	4571	0
fdct	40856	0.12	9	0	40856	0
fibcall	3064	0.02	0	0	3064	0
inssort	18167	0.08	0	0	18167	0
jcomplex	2586	0.03	0	0	2586	0
lcdnum	5507	0.10	11	6	4907	11
minmax	563	0.07	0	3	455	19
ndes	795425	4.57	2	1	791980	0
ns	130733	0.51	0	0	130733	0
nsichneu	119707	357.31	1424	623	118923	1

Table 3. Infeasible path analysis results

Analysis of multi-path programs. In Table 4, we show the result of multi-path analysis using all algorithms for loop bounds and infeasible paths. We have selected a number of benchmark programs for which it is possible to create multi-path input data, i.e., data

Program	#I	Orig.		LBI+IN+UN+EP+IP				
		Time	WCET	Time	+%	#FF	WCET	−%
crc	16	4.90	834159	6.65	36	56	833730	0
edn	40	2.29	1425085	3.86	69	41	1425085	0
fibcall	100	0.09	10134	0.13	44	4	10134	0
inssort	10^{93}	0.16	31163	0.17	6	7	18167	42
jcomplex	100	0.30	22020	0.39	30	19	9640	56
lcdnum	16	0.21	5507	0.36	71	71	3187	42
minmax	27	0.08	563	0.08	0	2	563	0
ns	2	6.09	130733	6.81	8	15	130733	0
nsichneu	91	36.88	119707	435.70	1081	65280	41303	65

Table 4. Results for multi-path programs

forcing the analysis to take many paths through the program. This input data is given to our tool as interval annotations for variables at certain program points. Column #I shows the number of input data combinations of the given input data, i.e., the number of single-path executions this analysis corresponds to. For `inssort`, this input data represents all possible inputs to the program, and the WCET therefore is the WCET for *all* executions of the program.

The other columns in the table show analysis time in seconds (Time) and resulting WCET with only basic (necessary) loop analysis. The last five columns show the result of the additional analyses (LBI+IN+UN+EP+IP), and have the same contents as in Table 3. The analyses used merging of states after function and loop termination (see Section 4). We did not use merging after if statements or loop iterations, since this erases some information for the recorders, and could yield less precision.

With a reasonable analysis time in most cases, we obtain up to 65% reduction of WCET estimates. As for the single-path case, the WCET estimates for different programs are improved by different analyses: LBI is beneficial for nested loops with iteration-dependent inner loop bounds, whereas infeasible path analyses improve the WCET estimates for programs with many if-statements. Clearly, a range of different analyses are necessary to find tight flow constraints for different kinds of programs. The long analysis time of `nsichneu` is mainly due to the calculation phase.

9 Conclusions and Future Work

We have shown that abstract execution is able to automatically derive loop bounds and infeasible paths for a set of WCET benchmarks. In particular, we were able to derive WCET estimates fully automatically for all the benchmarks. This is important, since the manual calculation of program flow constraints can be both tedious and error-prone. Our analyses for improved nested loop bounds and infeasible paths were in addition able to improve the resulting WCET estimate with up to 65%. Not surprisingly, the improvements

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE

were very dependent on the type of analysis and the control structure of the program.

The times for the analyses are reasonably short for most of our benchmarks, also when multi-path analysis is performed. For one program the calculation phase took a long time, mainly due to the large number of generated flow constraints.

The next step is to try out the algorithms on industrial real-time codes. In a previous case study [20], we were able to obtain considerably better WCET estimates for some industrial real-time code by adding infeasible path constraints by hand. We plan to run our infeasible path analyses on the same code to see if we can derive these constraints automatically.

Clearly, it is important to avoid generating unnecessarily many flow facts: we have seen a large increase in calculation time in one case, and in other cases the additional flow facts did not yield an improvement in the WCET estimate. Our current implementation has only some rudimentary mechanisms to avoid generating useless constraints. There are ways to improve this: for instance, when different infeasible path analyses are run together, they will often generate redundant constraints, which should be quite straightforward to eliminate. We plan to investigate this further.

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] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. Applying static WCET analysis to automotive communication software. In *Proc. 17th Euromicro Conference of Real-Time Systems, (ECRTS'05)*, July 2005.

[4] T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra. Exploiting branch constraints without exhaustive path enumeration. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis, (WCET'2005)*, pages 40–43, July 2005.

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

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

[7] O. Eriksson. Evaluation of static time analysis for CC systems. Master's thesis, Mälardalen University, Västerås, Sweden, Aug. 2005.

[8] 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.

[9] A. Ermedahl, F. Stappert, and J. Engblom. Clustered worst-case execution-time calculation. *IEEE Transaction on Computers*, 54(9):1104–1122, Sept 2005.

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

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

[12] J. Gustafsson, A. Ermedahl, and B. 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)*, Feb. 2005.

[13] 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*, May 2000.

[14] 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.

[15] N. Holsti, T. Långbacka, and S. Saarinen. Worst-case execution-time analysis for digital signal processors. In *Proc. EUSIPCO 2000 Conference (X European Signal Processing Conference)*, 2000.

[16] A. 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.

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

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

[19] 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.

[20] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz. Static WCET analysis of real-time task-oriented code in vehicle control systems. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, November 2006.

[21] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

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

Proceedings of the 27th IEEE International
Real-Time Systems Symposium (RTSS'06)
0-7695-2761-2/06 $20.00 © 2006 IEEE