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, and which is the basis of the methods below.
- 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...
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.

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 arithmetic. 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:

\[
\text{if} \ (x<0) \ \text{A else} \ \text{B; if} \ (x>2) \ \text{then} \ \text{C else} \ \text{D}
\]

Here, both true-branches for the if statements are in conflict\(^2\), and the corresponding path A-C can never be taken. This kind 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.


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\(^3\) nodes of which exactly one node must be executed for each iteration of the scope: typically, this subset consists of the header node of a well-structured loop, or the entry node of

---

\(^1\)A perfect flow analysis would solve the halting problem.

\(^2\)We assume that \(x\) is not updated in \(A\) and \(B\).

\(^3\)CFG = Control Flow Graph
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

\[
\text{loop} : [ ] : \#H < k
\]

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

\[
\text{loop} : <3..7> : B1 + \ldots + Bn < n
\]

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 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 “⊔” on the abstract domain of states, then the result is an abstract state safely representing all possible concrete states, but possibly with some loss of precision. Different strategies for merging will thus yield different tradeoffs between analysis time.
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 <-
          final_state(s'):
          final_states <-
        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.

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.σ 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 inputs to the analy-
ized program: thus, the algorithm is input-sensitive.

Each scope sc has a collector sc.c, which collects in-
formation 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. Typi-
ically, update points occur at new iterations, and scope
exits. The algorithm is asynchronous since that up-
dates may occur in different order, depending on when
abstract states to execute are drawn from the work-
list. 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 al-
gorithm 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 arbi-
trary 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:

\[(i, r, p, σ) \sqcup (i, r', p, σ') = (i, r \sqcup r', p, σ \sqcup σ')\]

5 Calculation of Flow Information

We now present our methods to calculate loop
bounds and infeasible paths. All the methods are vari-
ations of the abstract execution described in Section 4,
with different recorders, collectors, update points, and
update operations. The abstract state transition func-
tion ae also differs in its handling of recorders depend-
ing on which method is used. The methods can be
combined in any combination. At the end of the analy-
ysis, each collector is used to generate flow facts for its
scope. The resulting flow facts can be merged freely.

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

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 Fig-
ure 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 algo-


\[(i, r, p, σ) \sqcup (i, r', p, σ') = (i, r \sqcup r', p, σ \sqcup σ')\]
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

\[
\text{update}(r, (l, u)) = (\min(r, l), \max(r, u))
\]

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 \text{l\_outer}, containing, at some nesting level, an inner loop with scope \text{l\_inner}. The method works almost exactly as the analysis in Section 5.1 applied to \text{l\_outer}: the only difference is that the recorder is incremented at each iteration of \text{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 \text{l\_inner} = \text{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:

\[
\text{scope} : [ ] : \text{BBB2} = 0;
\]

specifying that basic block BBB2 is not executed in any execution of the scope \text{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:

\[
\text{loop} : [ ] : \text{BBB1} <= 7;
\]

specifying that basic block BBB1 is never executed more than 7 times in any execution of the scope \text{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 \( \bot \), 0, or 1, corresponding to “Not visited pair”, “Excluding pair”, and “Not excluding pair”, respectively. Initially, all elements are \( \bot \). The update points are immediately before new iterations of the scope, before 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:

\[ \text{scope : < > : (#BB33 + #BB57) < 2; } \]

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 the path \( \text{A-B-D-F-G-H-J} \) through the CFG. In this tree, the paths \( \text{A-C} \), \( \text{A-B-D-E} \) and \( \text{A-B-D-F-G-I} \) are marked as infeasible. Similarly, Figure 4(c) shows the tree for the path \( \text{A-B-D-E-G-H-J} \). Note that the path \( \text{A-C} \) 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 \( \Lambda \), a leaf \( \text{leaf}(n, I) \) or \( \text{leaf}(n, F) \), where \( n \) is a CFG node, \( I \) represents an infeasible path and \( F \) a feasible path, or an internal node \( \text{int}(n, t, t') \) 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:

\[ t \sqcup \Lambda = t \]
\[ \text{leaf}(n, I) \sqcup \text{leaf}(n, I) = \text{leaf}(n, I) \]
\[ \text{leaf}(n, F) \sqcup \text{leaf}(n, x) = \text{leaf}(n, F) \]
\[ \text{int}(n, t, t') \sqcup \text{leaf}(n, x) = \text{int}(n, t,t'), \quad x = I, F \]
\[ \text{int}(n, t_1, t_2) \sqcup \text{int}(n, t_2, t_3) = \text{int}(n, t_1 \sqcup t_2, t_3) \]

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:

\[ \text{scope : < > : (#BB33 + #BB57 + #BB82) < 3; } \]

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: \text{main}, \text{foo1}, \text{foo2} (the two calls to \text{foo}) and their corresponding loop scopes, \text{foo1L} and \text{foo2L}, and \text{bar} and its two nested loops, \text{barL} and \text{barLL}, as shown in Figure 6. We assume that neither \( x \) nor \( y \) is changed in the code.

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

1. Infeasible nodes: none.
2. Infeasible pairs: \( \text{B-C, B-E, D-E, G-I, and H-J} \). Note that the path "A-C" actually represents \( 2^2 = 4 \) paths through the CFG, for which it is a prefix.
3. Infeasible paths: \( \text{A-D-E, B-C-E, B-D-E, B-C-F, G-I, and H-J} \).

Due to given input data constraints we also detect:

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. SWEET accepts input data specifications for integers, floats, pointers, unstructured code, and recursion. 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 an automatic method 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.

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 benchmarks without floating point calculations, since ARM9 lacks hardware support for these; however, the analysis as such can handle floating point calculations. 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, and number of elements. 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.

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 an automatic method 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.

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 benchmarks without floating point calculations, since ARM9 lacks hardware support for these; however, the analysis as such can handle floating point calculations. 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, and number of elements. 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.

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 an automatic method 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.

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 benchmarks without floating point calculations, since ARM9 lacks hardware support for these; however, the analysis as such can handle floating point calculations. 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, and number of elements. 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 an automatic method 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.

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 benchmarks without floating point calculations, since ARM9 lacks hardware support for these; however, the analysis as such can handle floating point calculations. 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, and number of elements. 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 an automatic method 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.
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. The column +% shows the extra time in % needed for the LBI analysis, and −% 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), 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} ≈ 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.

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 want to show the results of our algorithms for more complex test cases, therefore we have selected a number of
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 nsichnus 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 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

<table>
<thead>
<tr>
<th>Program</th>
<th>WCET</th>
<th>orig.</th>
<th>Time</th>
<th>+%</th>
<th>#FF</th>
<th>WCET−%</th>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>1009</td>
<td>0.02</td>
<td>0</td>
<td>0</td>
<td>1009</td>
<td>0</td>
</tr>
<tr>
<td>cover</td>
<td>37128</td>
<td>5.57</td>
<td>53</td>
<td>562</td>
<td>63563</td>
<td>15</td>
</tr>
<tr>
<td>ccc</td>
<td>834159</td>
<td>1.22</td>
<td>3</td>
<td>36</td>
<td>830278</td>
<td>0</td>
</tr>
<tr>
<td>edn</td>
<td>1425085</td>
<td>1.04</td>
<td>8</td>
<td>17</td>
<td>1425085</td>
<td>0</td>
</tr>
<tr>
<td>fac</td>
<td>4571</td>
<td>0.02</td>
<td>0</td>
<td>0</td>
<td>4571</td>
<td>0</td>
</tr>
<tr>
<td>fdct</td>
<td>40856</td>
<td>0.11</td>
<td>17</td>
<td>2</td>
<td>40856</td>
<td>0</td>
</tr>
<tr>
<td>fibcall</td>
<td>3064</td>
<td>0.02</td>
<td>0</td>
<td>2</td>
<td>3064</td>
<td>0</td>
</tr>
<tr>
<td>inssort</td>
<td>18167</td>
<td>0.08</td>
<td>0</td>
<td>4</td>
<td>18167</td>
<td>0</td>
</tr>
<tr>
<td>jcomplex</td>
<td>7276</td>
<td>0.02</td>
<td>0</td>
<td>0</td>
<td>7276</td>
<td>0</td>
</tr>
<tr>
<td>lcdmn</td>
<td>563</td>
<td>0.03</td>
<td>0</td>
<td>6</td>
<td>563</td>
<td>0</td>
</tr>
<tr>
<td>minmax</td>
<td>563</td>
<td>0.07</td>
<td>0</td>
<td>15</td>
<td>563</td>
<td>0</td>
</tr>
<tr>
<td>ns</td>
<td>130733</td>
<td>0.85</td>
<td>6</td>
<td>1</td>
<td>130733</td>
<td>0</td>
</tr>
<tr>
<td>nsichneu</td>
<td>119707</td>
<td>23.95</td>
<td>4</td>
<td>871</td>
<td>57247</td>
<td>52</td>
</tr>
</tbody>
</table>

### Table 3. Infeasible path analysis results

---

<table>
<thead>
<tr>
<th>Program</th>
<th>WCET</th>
<th>orig.</th>
<th>Time</th>
<th>+%</th>
<th>#FF</th>
<th>WCET−%</th>
</tr>
</thead>
<tbody>
<tr>
<td>ba</td>
<td>1009</td>
<td>0.02</td>
<td>0</td>
<td>0</td>
<td>1009</td>
<td>0</td>
</tr>
<tr>
<td>cover</td>
<td>37128</td>
<td>5.57</td>
<td>53</td>
<td>562</td>
<td>63563</td>
<td>15</td>
</tr>
<tr>
<td>ccc</td>
<td>834159</td>
<td>1.22</td>
<td>3</td>
<td>36</td>
<td>830278</td>
<td>0</td>
</tr>
<tr>
<td>edn</td>
<td>1425085</td>
<td>1.04</td>
<td>8</td>
<td>17</td>
<td>1425085</td>
<td>0</td>
</tr>
<tr>
<td>fac</td>
<td>4571</td>
<td>0.02</td>
<td>0</td>
<td>0</td>
<td>4571</td>
<td>0</td>
</tr>
<tr>
<td>fdct</td>
<td>40856</td>
<td>0.11</td>
<td>17</td>
<td>2</td>
<td>40856</td>
<td>0</td>
</tr>
<tr>
<td>fibcall</td>
<td>3064</td>
<td>0.02</td>
<td>0</td>
<td>2</td>
<td>3064</td>
<td>0</td>
</tr>
<tr>
<td>inssort</td>
<td>18167</td>
<td>0.08</td>
<td>0</td>
<td>4</td>
<td>18167</td>
<td>0</td>
</tr>
<tr>
<td>jcomplex</td>
<td>7276</td>
<td>0.02</td>
<td>0</td>
<td>0</td>
<td>7276</td>
<td>0</td>
</tr>
<tr>
<td>lcdmn</td>
<td>563</td>
<td>0.03</td>
<td>0</td>
<td>6</td>
<td>563</td>
<td>0</td>
</tr>
<tr>
<td>minmax</td>
<td>563</td>
<td>0.07</td>
<td>0</td>
<td>15</td>
<td>563</td>
<td>0</td>
</tr>
<tr>
<td>ns</td>
<td>130733</td>
<td>0.85</td>
<td>6</td>
<td>1</td>
<td>130733</td>
<td>0</td>
</tr>
<tr>
<td>nsichneu</td>
<td>119707</td>
<td>23.95</td>
<td>4</td>
<td>871</td>
<td>57247</td>
<td>52</td>
</tr>
</tbody>
</table>

### Table 4. Results for multi-path programs
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 analyses 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