Simple Analysis of Partial Worst-case Execution Paths on General Control Flow Graphs*

Jan C. Kleinsorge
TU Dortmund
jan.kleinsorge@tu-dortmund.de

Heiko Falk
Ulm University
heiko.falk@uni-ulm.de

Peter Marwedel
TU Dortmund
peter.marwedel@tu-dortmund.de

ABSTRACT

One of the most important computations in static worst-case execution time analyses is the path analysis which computes the potentially most time-consuming execution path in a program. This is typically done either with an implicit path computation based on solving an integer linear program, or with explicit path computations directly on the program's control flow graph. The former approach is powerful and comparably simple to use but hard to extend and to combine with other program analyses due to its restriction to the linear equation model. The latter approaches are often restricted to well-structured graphs, suffer from inaccuracy or require non-trivial structural analyses or graph transformations upfront or during their computations.

In this paper, we propose a generalized computational model and a comprehensive explicit path analysis that operates on arbitrary directed control flow graphs. We propose simple and yet effective techniques to deal with unstructured control flows and complex flow fact models. The analysis does not require a control flow graph to be mutable, is non-recursive, fast, and provides the means to compute all worst-case paths from arbitrary source nodes. It is well suited for solving local problems and the computation of partial solutions, which is highly relevant for problems related to scheduling and execution modes alike.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*

General Terms

Reliability, Verification, Performance

Keywords

Worst-case Execution Time, Path Analysis, Static Analysis

*This work was partially supported by Deutsche Forschungsgesellschaft (DFG) under grant FA 1017/1-1 and EU COST Action IC1202: Timing Analysis on Code-Level (TACLe).

EMSOFT '13 09/29/2013, Montreal, Canada

1. INTRODUCTION

The *worst-case execution time* (WCET) of tasks is a critical metric in all hard real-time systems. An upper bound on the WCET can be estimated by static analysis. Input to such an analysis is typically a program in its binary form. For this, fine grained analyses can be performed as instructions, data and their locations are usually complete and statically known. Worst-case estimations of the execution time of individual basic blocks can then be derived from possible CPU states and the access patterns to memories and caches. Ultimately, this information is used to perform an estimation of the WCET of an entire task. The *worst-case path* (WCEP) problem is to find the most time-consuming path through the control flow graph (CFG) from its entry to its exit. Cycles in the control flow pose a particular problem since without any knowledge of an upper bound on loop iterations, the worst-case path length is unbounded. The required *loop bounds* can be determined analytically or need to be supplied manually. As such, it is vital that the control flow reconstruction is able to identify loops correctly and to feature simple means to (manually) annotate the control flow with *flow facts* (e.g., loop bounds, alternative or infeasible paths) where necessary.

The predominant approach to WCEP analysis is the implicit path enumeration technique (IPET) [12]. The worst-case path problem is formulated as an integer linear program (ILP) which allows to easily model arbitrarily structured control flow and flow facts in one monolithic model. A disadvantage is that it only yields the WCET from the entry to the exit of a task and neither directly exposes the WCEP nor any information on subpaths. It is not ideal for modeling problems related to scheduling, where detailed information of timing of interior program points is required (e.g. multitask, multicore), for partial analyses in general (e.g. to provide bounds for schedulability analysis) and for analyses that could be interleaved with the path analysis to increase their accuracy, specifically.

As opposed to that, explicit path analysis techniques are significantly more difficult to employ and no generally accepted computational model is used. These analyses depend on the availability of high-level structural information of the control flow since loops are typically processed one at a time. The rationale is that longest paths are easily computed on the acyclic control flows of the loop bodies. Loops are then processed in a recursive manner starting from the most deeply nested ones. This is unfortunate since contextual information for a loop (such as path infeasibility or hardware states for timing analysis) is only available after it has been fully processed. Also, in this model, flow facts are either

978-1-4799-1443-2/13/$31.00 ©2013 IEEE

only allowed to be simple (one iteration bound for the entire loop), or the algorithms become significantly more complex [8]. For multi-entry loops, loop nestings are ambiguous and preprocessing steps [18] are mandatory that enumerate all possible combinations or even require a modification of the CFG. This can cause redundancies in the computations, requires changes to already existing frameworks and potentially reduces the analysis accuracy, as flow facts now do not necessarily match the presumed program semantics anymore. Although rare, it is critical to be able to handle such cases in all generality. Examples are analysis of low-level code, state machines, co-routines, cooperative scheduling, etc..

In this paper, we propose a generalized computational model and a comprehensive explicit worst-case execution path analysis for the evaluation of complex (partial) control-flows. Our concrete contributions are: The analysis can directly be performed on arbitrarily structured and immutable control-flow graphs. The computational model is non-recursive with regard to loops, cleanly isolates problems related to graph structure and context/path-sensitivity from the actual analysis and only exposes a simple, locally restricted join/transfer propagation model, which is easy to extend and specifically allows for the easy interleaving with other existing analyses. By means of path-compression and the avoidance of redundancy where possible, it scales very well. It computes worst-case paths at basic block level from arbitrary source points to all other points, as opposed to being limited to entry-to-exit paths and allows for the specification of complex flow constraints beyond simple loop bounds. In particular, the fine-grained analysis results provide a basis for improved interference analyses in multitask and multicore scenarios.

The paper is structured as follows. In Sec. 2, we review related work. In Sec. 3, we discuss problems related to the structure of CFGs and propose a simple solution. Sec. 4 addresses the actual path analysis by introducing the technical foundation and by presenting our reference implementation. We trade a thorough formal analysis in favor of a practical evaluation in Sec. 5. In Sec. 6 the paper is concluded and future work is discussed.

2. RELATED WORK

Path analyses in the domain of WCET analysis can roughly be separated into ones based on IPET [12] and ones based on algorithms for the classic *single-source shortest paths* problem [6]. They are two approaches to solve the same problem [16].

IPET is predominant in its original form even in industrial WCET analysis solutions [19], although variations based on parametric ILP [3] have been proposed. For explicit path analyses, however, different approaches are known [8, 2, 5]. For IPET, its limited extensibility inhibits a combined analysis of paths and CPU or cache states [17], which potentially profit from this additional contextual information. As opposed to that, explicit path analyses allow to directly model problems beyond the mere longest path problem, so that an interleaving with other analyses becomes possible [15].

Explicit path analysis techniques typically suffer from their inherent limitation to reducible graphs (single-entry loops) [9] and therefore rely on preprocessing such as node splitting [10] or explicit enumeration of possible permutations of loop nests [18]. The implication is that either control flow representations must be modified or the analyses must be tailored towards the underlying approach. Another typical limitation is the restriction to simple loop bounds. Most

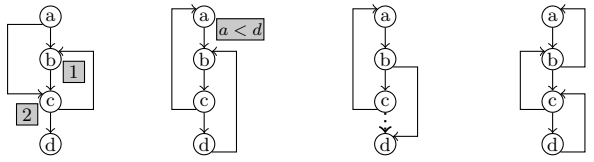


Figure 1: Examples for (a) ambiguous backward edges, (b) ambiguous scope nesting, (c) virtual exit edges and (d) virtual bottoms.

explicit path analyses perform their computations loop-wise or at a similarly coarse granularity. Mandatory worst-case bounds for loop iterations can then only be specified at this granularity as well [2], which can lead to over-estimations. An exception is [8], which comes at the price of a complex computational model. Except for [4] which is based on IPET, no approach addresses the problem of partial worst-case path analyses to the best of the authors' knowledge. Notably, the approach proposed in [2] is capable of recursively computing longest paths from a global entry node to all others but is restricted to single-entry, single-exit loops and to a simple flow fact model. A general overview of different path analysis techniques, in particular in the context of WCET analyses, is given in [19].

3. STRUCTURING CONTROL FLOW

In the following, we address problems related to control flow structure and loops. In particular we propose a very simple data structure and algorithms upon which a path analysis can be performed efficiently and non-recursively. In Sec. 3.1, we propose a data structure that models loop relations in a control flow and serves, along with the CFG, as input to the analysis. In Sec. 3.2, we present our approach to a non-recursive traversal of loop nests.

3.1 Scope Tree

If a CFG is reducible (all loops have a single entry) [13], we can directly construct a so-called *loop nesting forest* [14] by means of depth-first search (DFS) [6] to represent the relation of loops to one another and to define the membership of nodes to loops.

Let $G = (V, E)$ be a CFG. Given G is reducible, loop nesting forest construction relies on the classification of the CFG edges $E(G)$ into *forward* ($F(G)$) and *backward* ($B(G) = V(G) \setminus F(G)$) edges. Loops are then uniquely characterized by $B(G)$ and nesting relations are unambiguous.

The problem on irreducible graphs is that $B(G)$ is ambiguous, so that classic approaches to edge classification cannot be used directly. In Fig. 1(a), depending on whether a DFS starting in a visits b or c first, backward edges could be either (b, c) or (c, b). Worse yet, loop relations, such as depicted Fig. 1(b), can then also be ambiguous. Either loop $\{a, b, c\}$ or $\{b, c, d\}$ could be the nested loop, as far as program semantics are concerned.

Our approach is to avoid the problem of ambiguity altogether in the first place instead of being required to try out all possible combinations at some later time. The rationale is that such an enumeration makes little sense in the first place since the semantic model that was originally intended (by the developers) is unambiguous. Since a good understanding of program semantics is usually required to provide correct and

tight flow facts in the general case, we propose two simple extensions to flow facts beyond mere execution bounds.

First, we disambiguate the DFS by allowing flow facts to guide the algorithm in ambiguous cases. Such a *prenumbering* abstracts from the actual order in which nodes will be visited. In Fig. 1(a), only nodes b and c need to be prenumbered (grayed labels) such that the visitation order of a modified DFS corresponds to the numeric relation of the two numbers (b before c). For single-entry loops, these annotations are optional and therefore only required in rare cases.

Second, complex nesting relations as in Fig. 1(b) are unambiguously resolved by allowing explicit nesting relations as flow facts. In the figure, node a is supposed to belong to a loop that is nested in the one of node d (grayed label). Also here, this is optional and rather rare but is sufficient to model arbitrarily complex control flows.

For our analysis, a minimal data structure (similar to a loop nest forest) to maintain the structural information must be constructed. A *scope* denotes an *arbitrary* cyclic region in G. A *scope tree* $S = (V, E)$ is a directed graph, where each node $s \in V(S)$ represents a scope and each edge $(s, t) \in E(S)$ denotes that scope t encloses scope s. A *scope label* defined by $scope : V(G) \mapsto V(S)$ uniquely identifies the membership of a control flow node to a node of the scope tree. If a control flow node logically belongs to multiple scopes, its label is that of the innermost scope.

Per scope, we explicitly maintain four sets of nodes. Given that a scope s can now be unambiguously characterized by its backward edges $B_s \subseteq B(G)$, the (singleton) set *top* is defined as $top(s) = \{v|(u, v) \in B_s\}$. Analogously, we define $bottom(s) = \{u|(u, v) \in B_s\}$.

Let $\vec{G} = (V(G), F(G))$ denote the directed acyclic graph (DAG) of G. Let T denote the function that maps to the transitive closure of any directed graph. If $(u, v) \in E(T(\vec{G}))$, then node v can be reached from u.

Given the edge $(u, v) \in F(G)$, the labels $\bar{u} = scope(u)$, $\bar{v} = scope(v)$ with $\bar{u} \neq \bar{v}$. Then the set of entries are the nodes that can be reached from enclosing or neighboring scopes: $entry(\bar{v}) = \{v|(\bar{v}, \bar{u}) \in E(T(S)) \vee (\bar{u}, \bar{v}) \notin E(T(S))$. Analogously, the set of exits are the nodes that either reach an enclosing or neighboring scope: $exit(\bar{u}) = \{u|(\bar{u}, \bar{v}) \in E(T(S)) \vee (\bar{v}, \bar{u}) \notin E(T(S))$.

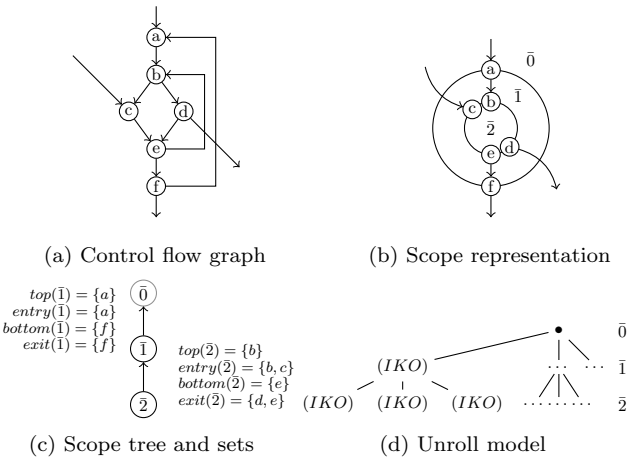
Fig. 2(a) shows a CFG of three scopes (outermost region and the two loops) that are explicitly depicted in Fig. 2(b) and labeled $\bar{0}$, $\bar{1}$ and $\bar{2}$, respectively. The corresponding scope tree is depicted in Fig. 2(c) along with the node sets.

Since we want to focus on the path analysis itself, we have to leave the algorithmic details for the prenumbering DFS and the scope tree construction aside.

3.2 Scope Order

For worst-case path computations, explicit path analysis techniques typically recursively descent into a loop nesting representation similar to our scope tree and compute longest paths one loop at a time starting from the innermost ones. Since the loop bodies – given nested loops have been successively replaced by (appropriately weighted) representative nodes – are acyclic, longest paths can be computed easily. The global worst-case path is then a comparatively unstructured concatenation of independent subpaths.

Although this approach seems quite generally applicable, it has several drawbacks. For multi-entry or multi-exit loops, it is not straightforward to replace a nested loop by some



(a) Control flow graph (b) Scope representation

(c) Scope tree and sets (d) Unroll model

Figure 2: Concepts of the computational model

representative. Even if measures have been taken earlier to be able to deal with multiple loop entries (cf. Sec. 2), multiple exits are still a problem[1]. Depending on the flow fact model, a loss of accuracy has to be accepted [2] or the control-flow representation becomes significantly more complex in addition to previous changes to the original CFG [10, 8]. Moreover, recursion implies that computations on nested loops are context-insensitive. Detection of unnecessary computations and early elimination of invalid data is not possible. For example, only once enclosing loops have been processed, path infeasibility of paths in subloops can be detected. This can also lead to much redundancy, which might not be an apparent problem for just the path analysis by itself. But once it is interleaved with other expensive static analyses [19], this can be significant.

We propose a simple non-recursive solution. In Fig. 2(a), if we remove the backward edges $((f, a), (e, b))$, a possible topological order of the nodes is (a, b, c, d, e, f). Replacing node labels by scope labels yields $(\bar{1}, \bar{2}, \bar{2}, \bar{2}, \bar{2}, \bar{1})$, or, without duplicates, $(\bar{1}, \bar{2}, \bar{1})$. This corresponds to the recursive traversal of the scope tree (from $\bar{1}$) in Fig. 2(c) and guarantees (here) that all nodes of scope $\bar{2}$ have been visited before all nodes of scope $\bar{1}$. Specifically, a scope is considered fully processed once all of its bottom nodes (cf. Sec. 3.1) have been visited since then all paths are known. Computations in $\bar{2}$ can therefore be context-sensitive and the worst-case path is composed in the "general direction" of control-flow in $\vec{G}$, not just for single loops.

In general, loops are not nested such that any arbitrary topological order yields the property that all nodes of nested loops have been visited before all nodes of enclosing loops. The rationale is that once all nodes of a scope are known, a (virtual) unrolling can be performed according to the given flow fact model.

The problem is twofold. In Fig. 1(c) (ignoring the dotted edge), node d could be visited before node c, which means the loop is not fully known once a node of the enclosing loop might already be dependent on that information.

Additionally, even if the topological order is unambiguous, for a graph layout such as depicted in Fig. 1(d), all nodes of a scope could have been visited before any nodes of its

[1]Multiple exits are far more common than multiple entries: e.g. error/exception handling

subscopes. Loops $\{b, c\}$ and $\{c, d\}$ are nested in $\{a, b\}$.

First, let $b^*(s)$ denote the set of bottoms of scope s and all of its subscopes. We call the set of "bottom-most" nodes of $b^*(s)$ the set of *virtual bottoms* and define it as:

$$vbottom(s) = \{u | u \in b^*(s), \forall v \in b^*(s) : (u, v) \notin E(T(\vec{G}))\}$$

In Fig. 1(d), the sets of virtual bottoms for all scopes are equal to $\{d\}$. All three scopes are fully known only once node d is visited.

Second, all scope bottoms must have been visited before any of the adjacent nodes of the exit nodes (cf. Fig. 1(c)). For $\vec{G}$, the *scope order* denotes a sequence of nodes $(u_1, \ldots, u_n)$ with $u_i \in V(\vec{G})$ such that for a scope s, a node $u_j \in exit(s)$ and a node $u_k \in vbottom(s)$ exists with $s = scope(u_j) = scope(u_k)$: $\forall u_l \in successors(u_j) : l > k$.

We can easily achieve the desired result by adding additional *virtual exit edges* from the (v)bottoms to the exit successors, as indicated by the dotted edge (c, d) in Fig. 1(c). A simple extension to an algorithm for topological sorting [6] suffices to achieve this without modifying the graph itself and at minimal costs.

Scope order simplifies the computation worst-case paths such that we can reason about worst-case paths in one control flow node by merely referencing its predecessors in $\vec{G}$.

4. PATH ANALYSIS

The objective of our path analysis is to compute the *latest execution time* (LET) for all nodes in the CFG. In the following Sec. 4.1, the overall computational model is presented. We briefly address our flow fact model in Sec. 4.2. Sec. 4.3 describes the algorithm itself.

4.1 Model of Computation

The basic principle of the LET computation is a virtual unrolling of all scopes. In such a – now – cycle-free control flow graph, the LET of the exit node of the control flow graph would be equal to the longest path from the root scope's top to its bottom.

To unroll a scope, we distinguish three phases for each scope: An *entry phase* I, a *kernel phase* K and an *exit phase* O. In the I phase, all paths from all entries to the bottoms can be taken. In the K phase, only paths from top to bottom can be taken. In the O phase, only paths from the top to the exits can be taken. A longest path through an unrolled scope is a concatenation of a single path of the I phase, as many paths of the K phase as possible according to some upper bound, and a single path of the O phase.

For each path of each phase of a parent scope, all subscopes have to be unrolled as well. Consider Fig. 2(d) which sketches the space of unroll phases for Fig. 2(b). For example, in scope $\bar{1}$, the three phases IKO have to be computed. For each such phase, three phases in scope $\bar{2}$ have to be computed as well, because for each path in $\bar{1}$, we also traverse $\bar{2}$. This is denoted by the edges in Fig. 2(d). All phase triples of one level in the figure refer to the same scope. But not all paths in the same phase of the same scope are feasible at all times. This depends on the current phases of the parent scopes.

For example, if scope $\bar{1}$ is in phase K, then the edges to and from scope $\bar{0}$ via the nodes c and d cannot have been taken, since only the paths from the top of $\bar{1}$ to its bottoms are possible. The observation is that phases of the same scope but under a different context potentially yield different results and thus must potentially be distinguished.

However, there usually exists a high degree of redundancy that we can exploit. For example, all K phases of scope $\bar{2}$ will yield identical paths regardless of its context, since the only paths that are feasible during this phase are the ones from the top to the bottoms. Also, the paths of the O phases are subpaths of the ones of the K phase. Depending on the entries and the context, I paths could also be redundant.

Reducible graphs are the special case where all phases of all scopes under all contexts have identical feasible paths.

We propose a virtual unrolling scheme that yields the least possible amount of redundancy and the ability to easily scale up to the most general case.

4.2 User Annotations

Besides the annotations for structural disambiguation (cf. Fig. 1(a), Fig. 1(b)) which are only needed in irreducible CFGs, we support a single numeric constant per node that bounds the maximal number of occurrences of this node on the unrolled path of a single scope. Fig. 3(a) gives an example of a scope with node a bounded by 1, node b by 2 and d by 0. This enables an accurate modeling of control flows since for multi-entry and multi-exit scopes in particular, bounds at loop-granularity (unroll factor) are insufficiently accurate. In addition, this allows for the specification of multiple, mutually exclusive longest paths in a scope and can easily be extended to deal with (global) infeasibility constraints. Since, for brevity, the paper focuses on the base algorithm only, we will not discuss global constraints in the following.

4.3 LET Computation

The algorithm is presented in two stages. In Sec. 4.3.1, the basic data structures, their construction, their relation to each other and the main pass of the algorithm are presented, which is sufficient to compute the LET of the exit. With the algorithm in Sec. 4.3.2, the LET of all interior nodes are computed.

4.3.1 Single Path LET

The path analysis solely depends on three input entities: DAG $\vec{G}$, scope tree S and flow bounds.

A node u is called a *flow anchor* if it has explicitly been annotated with a bound. The set of *flow bounds* is defined as $B = \{\ldots, (u, c)_{i \in \mathcal{B}}, \ldots\}$ where $u \in V(G)$ is the anchor and $c \in \mathbb{N}_o$ is its numeric value. We call the index set $\mathcal{B}$ the *flow bound index*.

Fig. 3(a) illustrates a scope with annotated flow bounds. Consequently, the set of flow bounds is $B = \{(a, 1)_0, (b, 2)_1\}$. The subscripted numerals denote the flow bound indices.

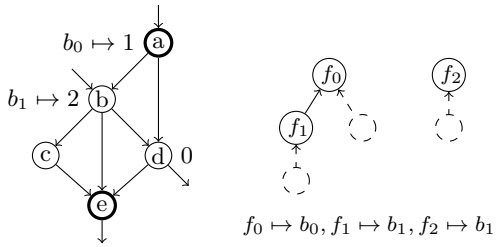
Also, each basic block represented by a control flow node has a specific WCET that is denoted by $cost : V(G) \mapsto \mathbb{N}_0$.

For the sake of simplicity, in the following, we use the tuple notation $(a, \_)$, where the underscore means "don't care" if an element is not relevant in the given context.

Simple Paths.

In a scope s, a simple path is defined as a path $p_{h,j} = (u_h, \ldots, u_j)$ such that its head $u_h \in entry(s)$ and $scope(u_j) = s$ for any node u_j. We define $length(p_{h,j}) = \sum_{h \le i \le j} cost(u_i)$. The trace of only the anchors A along such a path is:

$$A(p_{h,j}) = \begin{cases} A(p_{h,j-1}).(u_j) & \text{if } (u_j, \_) \in B \wedge h \le j \\ A(p_{h,j-1}) & \text{if } (u_j, \_) \notin B \wedge h \le j \\ () & \text{otherwise} \end{cases}$$



(a) Bounded scope (b) Respective flow trees

u	B	$V(F(a))$	$V(F(b))$	$P(u)$
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
a	$(a,1)_0$	$(b_0,1)_0$	$\emptyset$	$(a,f_0,1)$
b	$(a,1)_0,(b,2)_1$	$(b_0,1)_0,(b_1,2)_1$	$(b_1,1)_2$	$(a,f_1,2),(b,f_2,1)$
c	$(a,1)_0,(b,2)_1$	$(b_0,1)_0,(b_1,2)_1$	$(b_1,1)_2$	$(a,f_1,3),(b,f_2,2)$
d	$(a,1)_0,(b,2)_1$	$(b_0,1)_0,(b_1,2)_1$	$(b_1,1)_2$	$\emptyset$
e	$(a,1)_0,(b,2)_1$	$(b_0,1)_0,(b_1,2)_1$	$(b_1,1)_2$	$(a,f_1,4),(b,f_2,3)$

(c) Build-up of flow bounds, flow trees and path states

Figure 3: Interleaved construction of data structures

In Fig. 3(a), where a and e represent the top and the bottom of the scope, the complete set of traces for all paths $p_{a,e}$ is $\{(a,b),(a,d),(a,b,d)\}$ and for all paths $p_{b,e}$ it is $\{(b),(b,d)\}$.

A *flow tree* $F(u) = (V, E)$, where u is an entry node into a scope, is a compressed representation (a trie [11]) of each such a set of traces with common prefixes. Its set of nodes is defined as $V(F(u)) = \{\dots,(b,l)_{i\in\mathcal{F}(u)},\dots\}$ where $b \in \mathcal{B}$ refers to the corresponding flow bound and $l = \mathit{length}(p_{h,j})$ to the length of a simple path to the bound's anchor u_j. The index set of $V(F(u))$ is $\mathcal{F}(u)$. We refer to a path in such a flow tree as *flow trace*.

Fig. 3(b) shows the corresponding flow trees for the sets of traces. The nodes are labeled with the corresponding node indices, the mapping from flow tree nodes to flow bounds is shown beneath. Note that since the flow bound of CFG node d is zero-valued all paths through it are infeasible and we can ignore the traces that include this node (indicated by the dashed nodes). Note that the size of the flow trees is a function of the number of annotations not of the number of CFG nodes. Fig. 3(a) in turn depicts the mapping from flow bounds to their numeric values.

Finally, a *path state* represents the length of a simple path to a CFG node u_j within a scope. The set of path states is defined as $P(u_j) = \{\dots,(u_h,f,l),\dots\}$ where $u_h \in V(G)$ is the head node of a path $p_{h,j}$ (which is necessarily an entry), the index $f \in \mathcal{F}(u_h)$ refers to a node of a flow tree and $l = \mathit{length}(p_{h,j})$ is the length of that path.

The following relation of data structures holds: Let u_i be the most recent anchor on a path from u_h to u_j. If a path state $(u_h,f,l) \in P(u_j)$ exists, it references a unique flow tree node $(b,\_)_f \in V(F(u_h))$ which in turn references a flow bound $(u_i,\_)_b \in B$. In short, a path state summarizes a simple path from an entry to any node in the same scope, while it references a path in the flow tree that summarizes its history of visited flow bounds.

Two path states are called *comparable* iff their heads and flow tree nodes match. To compute the set of path states $P(u)$ in a node u is by propagation from preceding nodes in $\vec{G}$. Let $P_{pred}(u) = \cup_{v\in pred(u)} P(v)$ be the union of all predecessor states. Then $P(u)$ is the minimal set of predecessor path states that consist only of the incomparable path states of maximal length, adjusted by the costs of u:

$$P(u) = \{(u_h,f,l_i + \mathit{cost}(u)) | \forall (u_h,f,l_j) \in P_{pred}(u) : l_i \geq l_j\}$$

The general idea of this is to solve the k-shortest paths problem [7] by precomputing the minimal set of paths that are not comparable due to their unique flow traces. This prevents us from unnecessarily enumerating all paths later.

We compute flow bounds, flow trees and path states concurrently while traversing the control flow nodes in scope order. Fig. 3(c) illustrates how this is performed in general, assuming that every node has a cost of 1.

We start with an empty set of flow bounds, an empty forest of flow trees for either entry a ($F(a)$) and b ($F(b)$) and an empty set of path states ($P(u)$) and traverse the CFG nodes in Fig. 3(a) in scope order. Upon visiting a, an initial flow bound b_0 is initialized with an anchor value of a and a bound value of 1, according to the annotation in a. Concurrently, the flow tree $F(a)$ is initialized with a node f_0, referencing the flow bound b_0 and a distance value to this anchor of 1. Also an initial path state is emitted, which represents all paths starting in a, belonging to flow tree node f_0 and the same distance value as it's corresponding flow tree node.

Upon visiting b, a new flow bound b_1 and a new flow tree $F(b)$ are emitted. Flow tree $F(a)$ is extended by a new node f_1, which is now referenced by the path state for the entry a. A new path state is emitted that represents all paths from the entry b and which references $F(b)$.

Since node c is neither annotated nor an entry, merely the distance values of the path states are updated. Since node d is bounded by 0, the sets of flow bounds and flow tree nodes remain the same and we clear the set of path states

Finally, in e, the path states are joined by keeping the incomparable ones of maximal distance.

The result in e is a minimal set that represents simple paths through the scope and a path-compressed representation of the possible paths through the anchors. These paths are mutually dependent in that they refer to the very same flow bounds. This effectively leaves us with a network flow problem [6] that is constrained by the flow fact model.

Unrolling.

An *unroll path* of a scope is a concatenation of simple paths such that its first node u is the entry of the scope, its last node v is any node of the scope and no anchor occurs more often than its flow bound specifies. An unroll path $(u,\dots,v)$ is *maximal* if no other unroll path of greater length exists.

Recall from Sec. 4.1 that we conceptually distinguish three unroll phases I, K and O. For an unroll path to exist, the flow bounds must allow for at least one feasible simple path from an entry u to a bottom b (I path) and at least one feasible simple path from a top t to an exit v (O path) or, if no such two paths exist, at least a single feasible simple path from u to v. We refer to the length of a maximal unroll path from an entry to an exit as the maximal *scope weight*.

As an example, consider Fig. 3. For entry a (top) and exit e (bottom), we first make sure that a feasible I path is present. In the path states, only one path state with a root of a exists ($(a,f_1,4)$). Starting from flow tree node f_1, the

Algorithm 1 Unrolling

```
 1: function UnrollMax(I, K, O, D)
 2:     w ← w_max ← 0
 3:     for all i ← I with test(i) > 0 do
 4:         apply(i, 1)
 5:         w ← w + length(i)
 6:         for all o ← O with test(o) > 0 do
 7:             apply(o, 1)
 8:             w ← w + length(o)
 9:             for all k ← K do
10:                 n ← test(k)
11:                 apply(k, n)
12:                 w ← w + length(k) * n
13:             w_max ← max(w_max, w)
14:             reset_flow
15:             w ← 0
16:     if w_max = 0 then
17:         for all d ← D with test(d) > 0 do
18:             return length(d)
19:     return w_max
20: end function

21: function Unroll(s, u, v)
22:     B ← ∪_{b∈vbottom(s)} P(b)
23:     I ← {p|(h, f, l) = p ∈ B, h = u}
24:     K ← {p|(h, f, l) = p ∈ B, h ∈ top(s)}
25:     O ← {p|(h, f, l) = p ∈ P(v), h ∈ top(s)}
26:     D ← {p|(h, f, l) = p ∈ P(v), h = u}
27:     return UnrollMax(I, sort_len(K), O, sort_len(D))
28: end function
```

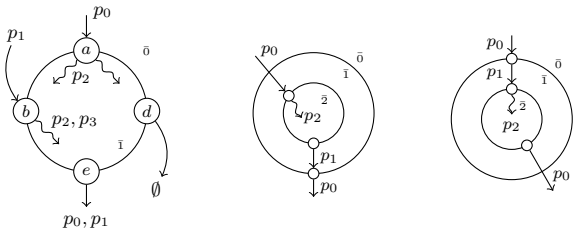
minimal flow bound along the path is obtained by traversing up the flow tree ("test") instead of traversing the actual path. Since the I path is taken once only, we now "apply" the flow by effectively decreasing the bound value on a to 0 and on b to 1[2]. Trying to reserve a K or O path for entry a will now fail since no feasible path exists anymore (due to the lowered flow bounds). There exists no feasible, unroll path for this entry. However, there exists a feasible direct path from a to e (I equals O path) with a weight of 4, which results from the path ($abce$).

For entry b (side-entry) and exit e (bottom), we test and apply an I path in the same manner. However, a feasible K path remains from a to e since we have not traversed node a before. Since this K path is also a legal O path, the unroll path is (bce).($abce$) with a weight of 7.

Alg. 1 sketches how we compute the maximal unroll paths, which slightly deviates from our example. We do not directly compute maximal unroll paths to all exits but generalize this computation for any node of the same scope.

The function *Unroll* is invoked with the scope label s, the scope entry u and a target node v (l. 21) and returns the length of the longest unroll path to node v for this entry. For multiple bottoms, we simply join all path states upfront (l. 22). In l. 23-26, we partition the path states for the I, K and O phases according to Sec. 4.1. The set D contains all path states from u to v. $UnrollMax$ is invoked with these sets, where the sets for the kernels and the direct paths are sorted by descending length (l. 27). In $UnrollMax$ (l. 1), we reserve one I path and one O path at a time and then maximize the length of the unroll path by reserving as many remaining K paths as possible. The function *test* traverses the flow traces in the flow tree for the given state and returns

[2]The principle is that of finding *residual paths* in the flow tree similar to the classical Ford-Fulkerson method [6].



(a) Restoring states (b) Far entry states (c) Far exit states

Figure 4: Handling external path states

the minimal flow bound value along its trace. If it returns 0, the path is infeasible. If it returns ∞ the unrolling is unbounded. In turn, *apply* reduces the flow bound values along the respective trace by the given amount[3]. For each pair of I and O paths, we dedicate all remaining flow n to the K paths in descending length to maximize the length of the unroll path (l. 12). Also, for each such pair we have to repeat the computation. Thus, the flow bounds must be reset to their original value (l. 14). If no feasible I and O paths could be found, we use the longest direct path from the entry to the target node (l. 16-18).

Computing unroll paths to all nodes yields a lot of redundancy. For example, in Fig. 3(a), for entry a, node e and the least recent (feasible) anchor b, it suffices to unroll the loop for b as the target node to directly derive the path lengths to c and e as well without further unrolling. Therefore, we effectively compute the longest unroll paths for all pairs of entries and anchors instead and only derive all other path lengths (including all exits) later.

Unrolling is the most expensive computation of the analysis. However, the complexity of computing all maximal weights for all pairs of source and target nodes depends primarily on the complexity of the flow fact model, to a lesser degree on the number of entries and exits but not on the complexity of the CFG itself, since only path summaries exist that are discriminated by their flow traces. Due to the flow tree, we explicitly avoid enumerating all possible path combinations, which is a frequent bottle-neck of path-based analysis techniques.

States and Scope Transitions.

If a path crosses one of its child scopes, its length is extended by the respective weight. Fig. 4(a) schematically illustrates the scope in Fig. 3(a) (without interior nodes), labeled $\bar{1}$. The path states p_0 and p_1 from some enclosing scope $\bar{0}$ are propagated to the entries a and b of scope $\bar{1}$.

Since we need to compute the simple paths and the scope weight of $\bar{1}$ first before continuing in $\bar{0}$, p_0 is replaced in a by a path state p_2. Analogously, p_1 is replaced by p_3 for the side entry b. The states p_2 and p_3 correspond to the path states in example Fig. 3(c). Once we visited all nodes of $\bar{1}$, its simple paths are known and we can unroll, restore p_0 and p_1 in e and update their lengths by respective weights of $\bar{1}$.

For a node $v \in V(\vec{G})$, a path state of a preceding node $u \in V(\vec{G})$ is called *external* iff it belongs to a parent scope. If it belongs to the same scope, it is *internal*. We guarantee that no other cases occur by design. In Fig. 4(a), state p_0 is external in node a. Generally, all external states are replaced by a representative internal state in each entry. Replaced

[3]We solve a variant of the MIN-COST flow problem [1] with node demand on a path-compressed network.

external states are *pending* until they are restored. Each new initial state implies a new flow tree and a corresponding (default) flow bound. To carry information from the outside into a scope (e.g. to implement global path constraints) initial states can be created as required at this point.

We call entries from and exits to non-adjacent scopes *far entries* and *far exits*, respectively. In Fig. 2(b), node c is a far entry and d is a far exit. We refer to path states that originate from a non-adjacent parent scope as *far states*.

Far entry states are handled just like all other external states. They are stored away as pending states and an initial path state is emitted for the subscope. In Fig. 4(b), the path state p_0, which is internal to scope $\bar{0}$, is propagated to the far entry of scope $\bar{2}$. State p_0 becomes pending and is replaced by p_2 in $\bar{2}$. However, in the exit of $\bar{2}$, restoring p_0 would be incorrect. We are supposed to continue the computations in scope $\bar{1}$, but p_0 represents a simple path in $\bar{0}$. We solve this by recognizing the far entry to $\bar{2}$ as yet another entry to $\bar{1}$. This dynamically extends the static set of entries we computed during scope tree construction.

The set of *virtual entries* denotes the set of all entries to the current scope and the set of all entries to its subscopes where an entry to a subscope is included iff its set of path states includes far states. Instead of restoring p_0 in $\bar{1}$, we add the entry of $\bar{2}$ to the virtual entries of $\bar{1}$, migrate p_0 into the set of pending states of $\bar{1}$ and create a path state p_1 that now represents the path to the exit of $\bar{1}$. Finally, p_0 is correctly restored when resuming computations $\bar{0}$.

Symmetrically, a similar problem exists for far exits, as illustrated in Fig. 4(c). The path states p_0 and p_1 enter the scopes $\bar{1}$ and $\bar{2}$ through their top entries. States p_1 and p_2 are created as usual. In the exit node of scope $\bar{2}$, state p_2 would be replaced with p_1 to resume a computation in $\bar{1}$. However, the target scope of the exit edge is $\bar{0}$. The path length represented by p_2 in the exit is only that of a simple path in $\bar{1}$ and the weight of $\bar{2}$. But he latest execution time is only obtained by also unrolling $\bar{1}$ before restoring p_0. We achieve this by dynamically extending the set of exits of $\bar{1}$.

The set of all exits of a current scope and the set of all far exits of its child scopes whose out-edges do not target another subscope of the current scope are called *virtual exits*. In the example, if the far exit becomes another exit of $\bar{1}$, then once $\bar{1}$ is resumed, p_2 is replaced by p_1 as usual. However, once computations in $\bar{0}$ are resumed, p_0 is restored including the scope weights of both subscopes.

Scope order and the non-recursive handling of states according to these rules yield the key to a simple join/transfer model. It is always safe to unconditionally access the path states in predecessor nodes to compute simple paths lengths and it enables the analysis to start and terminate in any node.

Primary Algorithm.

Alg. 2 shows the computation of the LET to the exit node of G and how to maintain intermediate results such that the LET to all nodes can be computed cheaply in a second pass. We chose a high degree of detail since the general ideas so far do not reflect the full set of ideas that would yield a fast and self-contained algorithm. In l. 1-5, some mappings are defined. The set *nbottom* counts how many (virtual) bottoms of a scope have already been visited, so that we know when we can unroll the scope.

Algorithm 2 Computation of sink LET

```
 1: nbottom : scope ↦ int
 2: ventry, vexit : scope ↦ node*
 3: maxlen : node ↦ node ↦ int
 4: pstate, estate : scope ↦ node ↦ path_state*
 5: state : node ↦ path_state*

 6: function ENTER(s, u)
 7:     (int, ext) ← join_min(∀v ∈ pred(u) : state(v))
 8:     state(u) ← int
 9:     if ext ≠ ∅ ∨ u = entry(G⃗) then
10:         while u ∈ top(s) do
11:             nbottom(s) ← |vbottom(s)|
12:             ventry ← entry(s); vexit ← exit(s)
13:             s ← parent(s)
14:         pstate(s, u) ← ext
15:         state(u) ← state(u) ∪ {(u, (u, ∞)_b, 0)_f, 0)}
16: end function

17: function ANNOTATE(u)
18:     if is annotated then
19:         if (n ← annotation(u)) > 0 then
20:             B ← B ∪ {(u, n)_b}
21:             for all (u', f', l') = p ∈ state(u) do
22:                 p ← (u', (b, l')_f, l')
23:         else
24:             state(u) ← ∅
25: end function

26: function RESTORE(s, v, w, l_max)
27:     s_p ← parent(s)
28:     vexit(s_p) ← vexit(s_p) ∪ far_exit(s, v)
29:     R ← ∅
30:     for all (_, _, l) = p ∈ pstate(s, w) do
31:         if ¬far(s, p) then
32:             R ← R ∪ {(_, _, l + l_max)}
33:         else
34:             ventry(s_p) ← ventry(s_p) ∪ {w}
35:             pstate(s_p, w) ← pstate(s_p, w) ∪ {p}
36:             R ← R ∪ {(w, ((w, ∞)_b, 0)_f, l_max)}
37:     return R
38: end function

39: function FINISH(s)
40:     for all a ∈ anchors(s) do
41:         for all v ∈ ventry(s) do
42:             maxlen(v, a) ← UNROLL(s, v, a)
43:     l_max ← 0
44:     R ← ∅
45:     for all v ∈ vexit(s) do
46:         for all w ∈ ventry(s) do
47:             for all (_, (b, l_f)_f, l_s) = p ∈ state(v) do
48:                 l ← maxlen(w, anchor(b)) + l_s − l_f
49:                 l_max ← max(l_max, l)
50:             (R, _) ← join_min(R, RESTORE(s, v, w, l_max))
51:         if estate(s, v) = ∅ then
52:             estate(s, v) ← R
53:         state(s, v) ← R
54: end function

55: function VISIT(s, u)
56:     ENTER(s, u)
57:     ∀p ∈ state(u) : length(p) ← length(p) + cost(u)
58:     ANNOTATE(u)
59:     while u ∈ vbottom(s) do
60:         if (nbottom(s) ← nbottom(s) − 1) = 0 then
61:             FINISH(s)
62:         s ← parent(s)
63: end function

64: foreach u in scope_order(V(G⃗)) : VISIT(scope(u), u)
```

The virtual entries and exits are denoted by *ventry* and *vexit*. The set *maxlen* holds the maximal scope weights for each anchor, discriminated by entry. We will later rely on this information to compute the longest absolute path to specific nodes. The pending states for each scope and each entry are stored in *pstates*. In *estate*, the original path states of the exit nodes are stored before an enclosing scope is resumed. We need them later for absolute path length computations. The set *state* holds the path states per node.

For the entire computation, each node is visited exactly once in scope order (l. 64). Thus, for the current node u and its scope s, we first call *Enter* (l. 56,6). The minimal set of path states (l. 7) is computed from the states of the preceding nodes where $join_{min}$ returns a pair that discriminates internal and external states. The internal states *int* are maintained (l. 8) but the external states *ext* denote an entry and need be replaced.

If u is an entry (l. 9), some initialization takes place. If u is a top entry, it could be shared among scopes. Thus, for each scope that is entered through this node, we count its bottoms and initialize the sets of virtual entries and exits with their actual entries and exits (l. 10-13). External states become pending for this entry (l. 14) and an initial path state is created that represents the path starting in u, references a flow tree node and has a length of 0. The flow tree node is initialized with a default flow bound and a distance of 0 (l. 15, written in-line for compactness). The set $state(u)$ now (l. 57) only consists of internal states and we can unconditionally update their length values.

Flow bounds are applied in *Annotate* (l. 58,17). The function *annotation* (l. 19) returns the user-supplied flow bound value. If it is feasible, we create a new flow bound (l. 20), and assign to each path state a new flow tree node that references the common bound (l. 21,22). This implicitly grows the flow trees.

Finish is invoked once all bottoms of a scope have been visited (l. 59-62), which in turn computes the maximal scope weight from each entry to all anchors (l. 40-42). *Unroll* is described in Alg. 1.

For each exit, we iterate over all entries to assemble all the (feasible) unroll paths to this exit (l. 45,46). We already computed the maximal lengths from each entry to each anchor. The distance from an anchor to the exit v directly be derived.

All states in v (l. 47) are iterated to obtain their respective most recent flow nodes (b, l_f). The maximal scope weight for this state is the distance to the anchor ($anchor(b)$) plus the distance from the anchor to v (l. 48). From all unroll lengths, we keep the longest path from entry w to exit v (l. 49).

The set (of external states) R holds all the pending states from all entries (l. 44,50). In *Restore* (l. 50,26), the states that became pending in entry w shall be restored in exit v. First, the set of virtual exits of the parent scope is extended (l. 28) by those exits v that are supposed to be virtual exits. If a state is not external, we only adjust the length according to the scope weight we just computed (l. 32). Otherwise, we extend the set of virtual entries of the parent scope (l. 34), export the far state into the set of pending states of the parent scope for the new virtual entry (l. 35) and "synthesize" a non-far state (l. 36) that shall be resumed instead.

Algorithm 3 Computation of LET to all nodes

```
 1: let : node ↦ int
 2: eoffset : scope ↦ node ↦ int

 3: function UpdateScope(s)
 4:     for all v ∈ ventry(s) do
 5:         off ← 0
 6:         if pstate(s, v) = ∅ then
 7:             ∀(_, _, l) = p ∈ pstate(s, v) : off ← max(off, l)
 8:         else
 9:             poff ← eoffset(parent(s))
10:             for all w ∈ ventry(parent(s)) do
11:                 for all (_, (l_f, b), l_s) = p ∈ pstate(s, v) do
12:                     l_a ← maxlen(w, anchor(b))
13:                     off ← max(off, poff(w) + l_a + l_s − l_f)
14:         eoffset(s, v) ← off
15: end function

16: function UpdateNode(s, u)
17:     if u ∈ exit(s) then
18:         state(s, u) ← estate(s, u)
19:     off ← 0
20:     for all (v, (l_f, b), l_s) = p ∈ state(s, u) do
21:         l_e ← eoffset(s, v)
22:         l_a ← maxlen(u, anchor(b))
23:         off ← max(off, l_e + l_a + l_s − l_f)
24:     let(u) ← off
25: end function

26: function Visit*(s, u)
27:     while u ∈ top(s) do
28:         q ← (q).(s)
29:         s ← parent(s)
30:     foreach i in reverse(q) : UpdateScope(i)
31:     UpdateNode(s, u)
32: end function

33: ∀s ∈ scopes : eoffset(s) ← 0
34: foreach u in scope_order(V(G⃗)) : Visit*(scope(u), u)
```

This state then represents the simple path that starts in the entry w and references the root of a new flow tree which refers to a default flow bound and its anchor w. The original states of (the innermost scope) in the exit node are saved for later use and replaced (l. 51-53).

4.3.2 All Paths LET

To compute the LET of each node, we have to take into account that each scope itself has a latest execution time we do not necessarily know yet. Alg. 3 shows its computation which is carried out after Alg. 2.

Two additional mappings are defined (l. 1,2). The set *let* will hold the LET of each node, and *eoffset* maintains for each scope and each entry the length of the longest path of the parent scopes to that entry. As previously, we visit the nodes in scope order (l. 34). In *Visit** (l. 26), if the current node u is a top node, we record the sharing scopes in the queue q from the innermost to the outermost and invoke *UpdateScope* which populates *eoffset* (l. 27-30). If the entry v has no pending states (l. 6), then it must belong to the root scope and the LET to it is just the maximal length of all simple paths (l. 7). Otherwise, the LET depends on the LET to the parent scope (l. 9) and the longest path from any entry of the parent scope (l. 10) to the entry v of the current scope s. The set $pstate(s, v)$ (l. 11) yields the states of the parent scope before they were replaced (cf. Alg. 2).

The LET for v (l. 13) is the maximal sum of the LET to w ($poff(w)$), the maximal unroll path length from w to the anchor (l_a) and the distance from that anchor to v ($l_s - l_f$).

Each individual node's LET is computed in *UpdateNode* (l. 31,16). If u is an exit node, then its original states have been replaced by pending states and have to be recovered (l. 18). The final, absolute path length (l. 20-24) is the maximal sum of the LET to the path head of each path state in the entry v (l_e), the maximal unroll path length from v to the anchor (l_a) and the distance of the anchor to u ($l_s - l_f$). The mapping let now defines each node's LET (l. 24).

5. EVALUATION

In the following, we evaluate the average performance of our path analysis (called *PAAN* in the following). Due to space limitations, we omit a formal analysis of worst-case complexity bounds. Instead, we provide experimental results for inputs that provide a good coverage of average- and corner-cases alike. All evaluations are carried out on CFGs constructed from randomly generated abstract syntax trees (AST) with annotations. The CFG ranges from 10 to approximately 60 000 nodes with 50 samples taken equally distributed. The AST is composed of four high-level language constructs IF, IFTHEN, WHILE and DOWHILE. Additional entries and exits to and from loops can be generated, as well as loop bounds and per-node WCET. Program semantics are not considered. We explicitly decided against evaluating "real world" benchmarks since they would be low in number (benchmarks must be flow fact annotated, thus the number applicable benchmark suites is limited) and would not provide the structural diversity to provide a satisfying coverage. Hard real-time benchmarks are also usually small in size and are therefore not well suited to demonstrate scalability properties of our algorithm.

For reference, we compare our results to that of the IPET path analysis. The reason is threefold. First, it is the only technique whose input can directly be derived from graphs of arbitrary structure and is self-contained in that is does not require any significant preprocessing similar to our approach and unlike other approaches. Second, it is the predominant approach to this problem. Third, most other path analyses are special cases of our approach. PAAN and IPET compute identical results on identical graphs (with one exception we point out below). However, IPET is only capable of computing the WCET of the globally longest path whereas we compute all longest paths to all nodes.

The experiments are carried out on a single core of an *Intel Xeon X3220* (2.4GHz) CPU. We measure the accumulated CPU time of all phases of our path analysis including the control flow reconstruction (which we excluded from this paper). For IPET, the construction of the linear equations is included. The IPET ILP is solved using *CPLEX* (12.4) with default arguments.

In Fig. 5, we compare the computation times for inputs that are quite "typically" found in real-time benchmarks. The graphs are reducible and are generated with probabilities P(IF) = 0.1, P(IFELSE) = 0.2, P(WHILE) = 0.3, P(DOWHILE) = 0.4, a probability for additional flow bounds of 0.1, a maximal loop depth of 3 and a single flow bound per loop. IPET (STD,WCET) denotes the time required to compute the worst-case path length from the entry to the exit. As compared to that, PAAN (STD,WCET) computes the same value with our proposed analysis. As opposed to that, PAAN

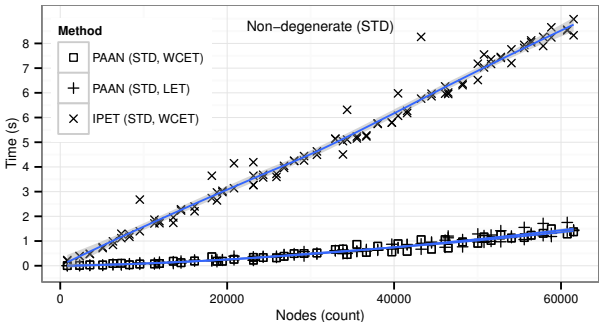


Figure 5: Runtimes on non-degenerated graphs

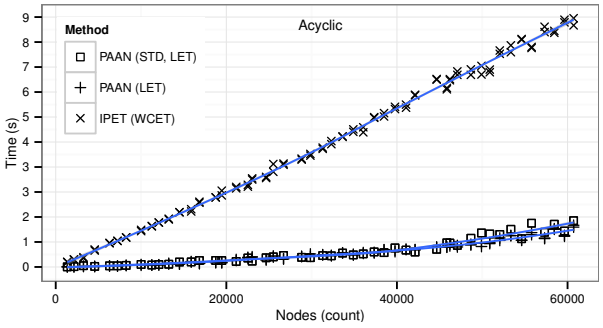


Figure 6: Runtimes on acyclic graphs

(STD,LET) computes the latest execution times of *all* nodes instead of just the single WCET of the global exit node.

The first observation is that PAAN performs and scales significantly better than the IPET, even if we compute the WCET to all nodes. It is also notable that it shows significantly less variance in its time consumption.

The time consumption for purely acyclic control flow (P(WHILE) = P(DOWHILE) = 0) is depicted in Fig. 6. We show the PAAN (STD, LET) results from Fig. 5 for reference and solve PAAN (LET) and IPET (WCET) for this specific graph type. IPET shows a much smaller variance. The primary insight however is that the curve for PAAN (LET) is practically identical to PAAN (STD, LET) and shows that the complexity of PAAN only marginally depends on the number of loops.

Due to the unrolling, PAAN could potentially be sensitive to graphs that are excessive regarding their number of loops, entries to loops and loop bound specifications. Therefore, we compare the time consumption for degenerated inputs. In the following, the probabilities are identical to the standard case except for the features we add for investigation.

Fig. 7 depicts the results for a parameterization of P(WHILE) = 0.3, P(DOWHILE) = 0.7 and a maximal nesting depth of 10. In this case, IPET shows to be less predictable. The high loop count has no significant effect on PAAN.

To summarize the figures so far, PAAN scales largely independently of the graph structure itself. It is now worthwhile to investigate extreme cases of irreducibility and flow bounds.

Fig. 8 shows the sampling for cyclic graphs with loops having entries to and exits from loops at practically all nodes (the exclusion of some nodes stems from the fact that nodes are restricted to an out-degree of 2 in the randomizer).

Nonetheless, we can now observe a significant impact for some of the inputs for PAAN as well as IPET, and a noticeable deviation from the reference (PAAN (STD, LET)). The grayed area denotes the confidence interval. The reason for

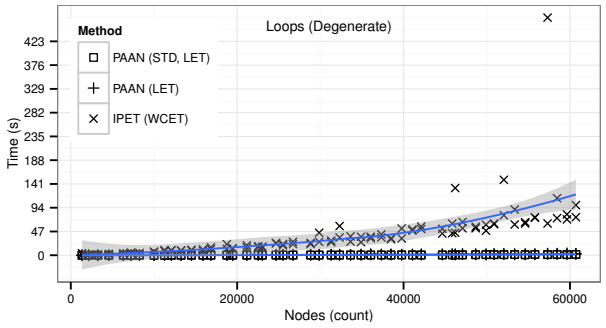


Figure 7: Runtimes with high loop counts

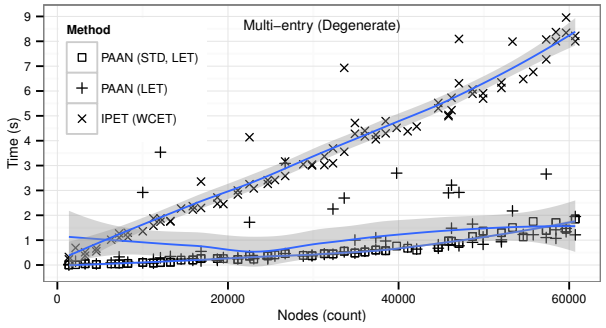


Figure 8: Runtimes on graphs with high entry and exit counts

the excessively large time consumption for PAAN in some cases is due to the control flow reconstruction (cf. Sec. 3.1). On average, scalability for PAAN is still very good.

As opposed to that, Fig. 9 shows results for control flow graphs where every single node is flow bounded. Since every flow bound potentially doubles the number of states, this is an approximation for the worst-case number of path states.

Again, IPET is comparably slow and yields a high variance. Comparing PAAN with its reference PAAN (STD, LET) does not show a noticeable difference on average. The reason for this is that exponential growth in the state space is always limited by scopes and for the root scope, we do not need to split states. Moreover, since all paths pass through the same bottom nodes, the flow bounds are quickly reached. PAAN and IPET do not necessarily compute identical results in this case since the semantics of both (randomized) flow fact models differ. With better random or careful manual annotations, identical results can be obtained in these cases.

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed a general and self-contained method for the computation of worst-case times on general control flow graphs. We discussed several general problems of control flow structure and proposed simple yet effective solutions to each of them and proposed a generalized computational model. The proposed path analysis is fast, simple, non-recursive, without structural restrictions, and allows for a fine grained flow fact model for accurate WCET estimations. It is the only approach that can directly be used to compute partial worst-case paths on general and immutable CFGs.

In the future, we will investigate ways to extend our flow fact model to global constraints and parametric analyses. Since the computational model itself is not restricted to worst-case time problems but can be easily interleaved with

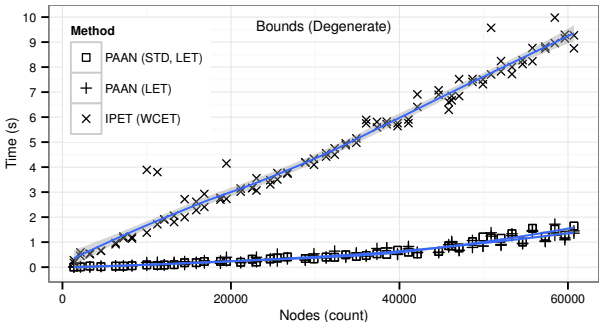


Figure 9: Runtimes on graphs with all nodes bounded

or extended by other analyses, we will investigate new strategies for a much more accurate handling of a multitude of open problems that arise in the context of path analysis, multitasking and multicore systems.

7. REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: Theory, Algorithms, and Applications*. Prentice-Hall, 1993.

[2] E. Althaus, S. Altmeyer, and R. Naujoks. Precise and Efficient Parametric Path Analysis. In *Proc. of LCETS*, 2011.

[3] S. Altmeyer, C. Hümbert, B. Lisper, and R. Wilhelm. Parametric Timing Analysis for Complex Architectures. In *Proc. of RTCSA*, 2008.

[4] S. Altmeyer, C. Maiza-Burguière, and R. Wilhelm. Computing the Maximum Blocking Time for Scheduling with Deferred Preemption. In *Proc. of STFSSD*, 2009.

[5] A. Colin and G. Bernat. Scope-Tree: A Program Representation for Symbolic Worst-Case Execution Time Analysis. In *Proc. of ECRTS*, 2002.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[7] D. Eppstein. Finding the K Shortest Paths. In *Proc. of FOCS*, 1994.

[8] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[9] P. Havlak. Nesting of Reducible and Irreducible Loops. *TOPLAS*, 19(4):557–567, 1997.

[10] J. Janssen and H. Corporaal. Making Graphs Reducible with Controlled Node Splitting. *TOPLAS*, 19(6):1031–1052, 1997.

[11] D. E. Knuth. *The Art of Computer Programming, Vol. 3*. Addison Wesley, 1998.

[12] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. In *Proc. of DAC*, 1995.

[13] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[14] G. Ramalingam. On Loops, Dominators, and Dominance Frontiers. *TOPLAS*, 35(5):233–241, 2002.

[15] F. Stappert, A. Ermedahl, and J. Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proc. of CASES*, 2001.

[16] R. E. Tarjan. A Unified Approach to Path Problems. *J. ACM*, 28(3):577–593, 1981.

[17] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *RTS*, 18(2/3):157–179, 2000.

[18] S. Unger and F. Mueller. Handling Irreducible Loops: Optimized Node Splitting versus DJ-Graphs. *ACM Trans. Program. Lang. Syst.*, 24(4):299–333, July 2002.

[19] R. Wilhelm, J. Engblom, et al. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. *TECS*, 7(3):36:1–36:53, 2008.