Computing Maximum Blocking Times with Explicit Path Analysis under Non-local Flow Bounds*

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

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

ABSTRACT

Worst-case time (WCET) analyses for single tasks are well established and their results ultimately serve the purpose of providing execution time parameters for schedulability analyses. Besides WCET analysis, an important problem is maximum blocking time (MBT) analysis which is essential in deferred preemption schedules for the selection of preemption points. Among the most pressing problems in this context is the need for good path analyses, which are a fundamental bottleneck for selecting these points. Current state of the art relies on ILP-based or severely constrained explicit path analyses, both of which are unsatisfactory in general.

In this paper, we propose a general explicit path analysis to compute maximum blocking times, specifically for scheduling policies with deferred preemption. The proposal improves the current state of the art significantly for both WCET and MBT analysis, as it is efficient, accurate, easily extensible and computes path lengths between all program points, without imposing any artificial constraints, and under a general flow bound model, unmatched by other existing explicit path analyses, while significantly outperforming the ILP-based approach. To the best of the authors' knowledge, no explicit path analysis for MBT has been proposed yet.

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 EU COST Action IC1202: Timing Analysis on Code-Level (TACLe) and by the German Research Foundation (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A3.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'14 October 12 - 17 2014, New Delhi, India
Copyright 2014 ACM 978-1-4503-3052-7/14/10...$15.00.
http://dx.doi.org/10.1145/2656045.2656051

1. INTRODUCTION

In embedded software systems, timing analysis and feasibility analysis of multi-task scheduling are key elements to reason about its timing behavior. For hard real-time systems, static *worst-case execution time* (WCET) analysis can provide safe upper time bounds for the uninterrupted execution of separate tasks. Schedulability analysis in turn asserts the applicability of a scheduling policy under given processing resources that are reflected by WCET.

Task preemptions cause task interruptions that are not reflected by the WCET of an isolated task alone. Direct and indirect context switch costs have to be taken into account. Direct costs are ones that result, for example, from saving and restoring execution context and from disruption of pipelined execution. The latter are caused by content changes to system caches due to execution of other tasks. These *cache-related preemption delays* (CRPD) are indirect because they do not necessarily occur right at the preemption point, after a task is resumed, but have a potential effect on the execution time at a later program point, when cache contents are actually requested. They are highly variable and potentially vastly dominate overall preemptions costs. Direct costs, on the other hand, are comparably small and can often be safely bounded with a constant.

CRPD can be bounded by statically analyzing access patterns of the cache by the preempting tasks and the reuse pattern by the preempted task. However, in *fully preemptive scheduling*, neither points in control flow nor points in time can be accurately determined at which preemptions occur. This means only worst-case assumptions can be made about the number of preemptions and the CRPD, which leads to significant overestimations. Overall, static timing predictability is low. The most important advantage is that fully preemptible tasks yield low average response times for higher priority tasks.

Non-preemptive scheduling represents the other extreme. On the one hand, timing predictability is very high, since the CRPD, as a major source of inaccuracy, does not have to be taken into account. On the other hand, some task sets that have been schedulable under fully preemptive scheduling potentially have no feasible schedule under this policy. The reason for this is the blocking time that lower priority tasks now impose on tasks of higher priority ready to execute.

The compromise is to employ scheduling policies with only limited preemption — so called *deferred preemption scheduling* — which either feature floating or fixed non-preemptive regions. If preemptions are time-triggered, then it is in general not possible to know the exact execution

context in which a preemption occurs. Hence the name *floating region*. Since strict timing guarantees can be given on the length of such a region, existing schedulability tests can directly be applied. The downside is, again, the lack of accuracy for the CRPD. *Fixed regions* are established by adding explicit preemption points to the program code. Although program context can now be determined much more accurately at the point of preemption, it is hard to determine the point in time of the preemption, and thus, the length of these regions. Moreover, to significantly reduce the actual and estimated costs of fixed regions, preemption points have to be chosen carefully.

To support fixed preemptive regions, *maximum blocking time* (MBT) analysis is required which computes the longest paths between potential preemption points. The goal is not to find any longest path between two program points, but the longest path to a specific program point, such that it only occurs once on this path — the most distant preemption points. In general, the lack of tools to tighten the relation between execution time and program context is a major obstacle in enhancing the accuracy of static multi-task analyses. In this context, WCET analysis and MBT analysis allow to map program points onto execution time and vice versa. Improving the accuracy of either potentially enhances worst-case analysis of all three kinds of preemption policies mentioned.

In this paper, we propose a general and efficient algorithm for MBT analysis, which advances the state of the art in explicit path analysis to facilitate deferred preemption schedules. It is designed not only to serve the mere theoretic purpose to enable such analyses at all, it is also technically simple. We extend an existing framework and provide a formal basis previously missing in its original proposal. Our analysis can be carried out on general control flow graphs, or subgraphs thereof, without any structural restrictions. Therefore, it enables not only the analysis of MBT from dedicated program points, such as task entries and exits, but supports the analysis from and to arbitrary program points. This is required for the selection of appropriate fixed preemption points in the first place, and it enables the consideration of execution modes and local reevaluation which directly supports computational modularity. It features an expressive flow bound model, and has specifically been designed to perform its computations by means of a simple data flow model, which simplifies extension and allows to interleave path computations with other analyses — such as static cache analysis — to achieve a higher degree of accuracy than it is currently possible with existing methods. To the best of the authors' knowledge, we are the first to propose a flow bound model for explicit path analysis that supports exact global bounds and the first to propose an algorithm for explicit MBT analysis.

In the following, related work is addressed in Sec. 2. The main body of the paper consists of Sec. 3, which addresses the foundations of path analysis, defines our computational model, proposes an implementation, introduces a general flow bound model and proposes an efficient encoding for MBT computations. An evaluation is provided in Sec. 4. We conclude the paper in Sec. 5 and provide an outlook.

2. RELATED WORK

Deferred preemption enhances predictability, simplifying static analysis [7]. In particular, fixed non-preemptive regions simplify CRPD analysis [12, 4]. Therefore, the authors of [6] address the problem of selecting optimal fixed preemption points among a set of predefined candidate locations, however, only on trivial control-flows. In [3], the problem of computing the MBT, given a set of fixed preemption points on general control flow graphs, is addressed. This is a precondition for the applicability of schedulability tests in the first place. Their approach is derived from the *implicit path enumeration technique* [13] (IPET), which is an *integer linear programming* (ILP) model for the computation of worst-case execution path lengths in single-task WCET analysis. IPET is unsuitable for WCET analysis beyond simple single task, entry-to-exit analysis since its constraint model is based on defining relations between program points, such that a computation on just subgraphs is in general not possible without generating tailored models for each case. Worse, problems are encoded in linear equations which severely hamper its flexibility and extensibility. These traits specifically impede improvements in system-level analysis, where a fine-grained mapping of state and time is of particular interest. The key advantage of IPET is its simplicity, in particular in the face of general control flow graphs and complex constraint models. Explicit path analyses [16] have not evolved significantly beyond simple analyses under restrictive assumptions. Some progress has been made in symbolic path-analysis [2, 5, 8, 10], which did not, however, tackle the fundamental restrictions that explicit analyses typically share, such as graph reducibility, context-insensitivity of paths and trivial constraint models, which make them inferior to IPET. Our explicit analysis proposed in [11] advances the state of the art by proposing a basic framework for general control-flow graphs and fine-grained loop-local flow constraints that allows the computation of worst-case path within arbitrarily structured tasks. To the best of the authors' knowledge, the only MBT analysis is based on IPET [3]. The MBT analysis is a derivative work of [11].

3. COMPUTING MBT

In this section, we address the problem of computing MBT. In Sec. 3.1 the formal foundation of worst-case path analysis is provided, followed by the concrete problem definition of MBT analysis in Sec. 3.2. We propose a model of computation for irreducible CFG in Sec. 3.3 and Sec. 3.4. In Sec. 3.5, we propose a general implementation for worst-case path analysis and details on the specific requirements of MBT analysis.

3.1 Environment

We first define the environment in which path analyses are carried out. A *control-flow graph* (CFG) $G = (V, E)$ is a connected digraph where its nodes V represent basic blocks[1] of a program and the edges $E \subseteq V \times V$ the structurally possible transfer of control between nodes. For convenience, we will use the notation $G(V)$ or $G(E)$ to refer to the respective sets of a specific graph. A CFG has a unique entry node s_G of in-degree 0 and a unique exit node of out-degree 0.

A *path* π is a sequence of edges and π^+ denotes its transitive closure. Node u *reaches* node v ($u \rightsquigarrow v$) if $\exists \pi : (u, v) \in \pi^+$.

A node u is said to *dominate* node v (u dom v), if all paths from s_G to v pass through u. A *depth-first search* partitions E into forward edges F and backward edges B such that

[1] A (maximal) sequence of instructions only entered from the first and left from the last one.

$\vec{G} = (V, F)$ is an *directed acyclic graph* (*DAG*).

A CFG is called *reducible*, if B is unambiguous. In such a graph, for each maximal set of backward edges $\{(b_0, h), \dots\}$, a *loop* $L_h \subseteq V$ contains all nodes that can reach any node b_i without going through node h ($\forall u \in L_h \colon h \text{ dom } u$). We call h the *head* and b_i the *bottoms* of a loop. A loop with header h is *nested* within another loop L, if $h \in L$. This partial order induces a *loop-nesting forest* — a set of disconnected trees denoting nesting relations, where every node represents a loop, uniquely identified by its head.

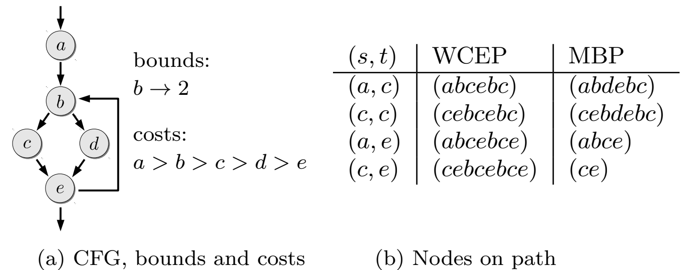
Reducible graphs are also referred to as "well-structured" or having "single-entry regions". As matter of fact, many concepts and techniques to reason about control- and data-flow in compilers depend on this particular property for reasons of simplicity and efficiency: Loops can be identified unambiguously and every loop without its backward edges forms a DAG with a unique entry node. In particular path problems profit from this property, since shortest as well as longest paths can be easily computed [2, 10, 14, 15]. In practice, most control-flow graphs are composed of single-entry regions since they are ultimately derivatives of some high-level program representation in which all constructs are single-entry. Moreover, every irreducible graph can be transformed into a reducible one by *node splitting* [9].

This is a reasonable justification for the restriction to just reducible graphs in optimization passes in compilers, whose purpose is to transform their input anyway. In worst-case path analyses, it is not. To obtain safe and accurate static analysis results to reason about program timing behavior, it is necessary to not only know the program semantics, but its actual interaction with the execution environment at the lowest possible level. In practice, control-flow is recovered from a program binary. The resulting CFG might be irreducible due to the use of "goto"-statements in the high-level program (state-machines, error-handling, co-routines, etc.), partial implementation in some low-level language ("inline assembly") or optimizations performed by the compiler.

Although node splitting could be performed here as well, it has severe drawbacks. First, the representation cannot be *immutable*, which has practical implications for its efficiency as a representation as well as computations on it (concurrency) and it is notorious for causing an exponential growth in graph size [9]. Second, although abstract program semantics do not change, the structural mapping from the actual program to the representation for the analysis is invalidated. This is critical as often manual annotations are required to support static analysis and different representations (physical and analysis model) become a source of complexity. Control-flow reconstruction is beyond the scope of this paper.

We now introduce the data structures to represent irreducible control-flow and assume them given. A *scope tree* [11] $\mathring{G} = (\mathring{V}, \mathring{E})$ is a generalization of a loop-nesting forest and represents the nesting relations $\mathring{E}$ of *scopes* $\mathring{V}$ — a generalization of loops. If we assume a given classification of edges in G in forward and backward edges, then a scope representing a loop denotes the set $S \subseteq V(\vec{G})$ such that for a (not necessarily maximal[2]) set of backward edges $\{(b_0, h), \dots\}$, each node $u \in S$ can be reached by its head h and which can reach its bottoms b_i in $\vec{G}$. A scope with head h is nested within another scope S with head h' if $h \in S \wedge h \neq h'$. If $h = h'$, nesting relation is only defined by $\mathring{E}(\mathring{G})$. The root of a scope tree denotes an entire program. As such, the outermost scope never represents a loop.

[2]It is not unusual for originally distinct loops to share the same head in a low-level representation. Intermediate representations in compilers would typically maintain empty head basic blocks (pre-headers) to facilitate loop identification.



(s,t)	WCEP	MBP
(a,c)	$(abcebc)$	$(abdebc)$
(c,c)	$(cebcebc)$	$(cebdebc)$
(a,e)	$(abcebce)$	$(abce)$
(c,e)	$(cebcebce)$	(ce)

(a) CFG, bounds and costs (b) Nodes on path

Figure 1: Examples relating WCEP and MBP

The function $\gamma(u) \in \mathring{V}$ denotes scope membership of a CFG node u, such that the function always maps to the innermost enclosing scope. For a node $\mathring{s}$, let $\text{par}(\mathring{s}) \in \mathring{V}$ denote the *parent* of node $\mathring{s}$ and $par^+(\mathring{s})$ denote the set of all ancestors. Then for an edge $(u, v) \in E(G)$ and $\gamma(u) \neq \gamma(v)$, node v is an *entry* to a scope, if $v \notin \text{par}^+(u)$. Analogously, node u is an exit, if $u \notin \text{par}^+(v)$. Entries and exits of the root scope coincide with those of the CFG.

The rationale for scopes is that the information which can be inferred from only the control-flow graph is insufficient for proper structural reconstruction. Scope trees make them explicit without imposing any restrictions on the original control-flow. In the following, we will informally refer to loops as synonyms for cyclic scopes and not in the formal sense as defined above.

An *iteration* of a scope is a path that starts in an entry and ends in either an exit or a bottom without passing through any backward edge. An *instance* of a scope is a path from an entry to an exit of a scope, composed of iterations.

WCET is the length of a *worst-case execution path* (WCEP) in a control-flow graph. Its length is determined by *node weights*, which is the WCET of individual basic blocks, and by *flow bounds* that limit the occurrence of these nodes on a path, so that its length is *bounded*. If a path cannot be realized, either because it is structurally impossible or due to flow bounds, it is *infeasible*. A *loop bound* is a flow bound that only constrains the head[3] of a loop, and is the simplest but most inaccurate constraint to bound path lengths.

3.2 Problem Statement

WCET and MBT are tightly related. Informally, a WCEP is a longest feasible and finite path from a source s to a sink t. A *maximally blocking path* (MBP) is a WCEP, with the additional constraint that t is not allowed to be on the path except as its end. The rationale is that the worst-case blocking behavior for tasks with explicit preemption points occurs for the longest path that does not include any preemption point.

As an example, consider Fig. 1(a) which depicts a CFG with a single flow bound that restricts the number of occurrences of node b on any path to at most twice, and whose nodes have execution costs in the given order. Accordingly, Fig. 1(b) lists the respective nodes on the WCEP and MBP, respectively, given explicit source and sink nodes.

[3]Alternatively a single bottom.

WCEP and MBP are to be interpreted slightly differently for a given CFG. A WCEP accounts for the longest path from the source node including its costs to the sink node excluding its costs. The rationale is that we are interested in the longest path until a program point is reached. For MBP, the source node is the successor of the basic block triggering the preemption. Technically, a preemption point is an unconditional transfer of control out of a task. By definition, it terminates a basic block in a CFG and execution can only resume in an adjacent basic block. The costs of a sink node are not accounted for, but can be trivially added after the analysis. We made this choice to maintain symmetry with the WCET problem. In the following, we always implicitly assume a virtual global exit node of zero costs that belongs to the root scope.

Two problems are of particular interest in MBT analysis:

1. Given a set of preemption points, compute the MBT (*MBT-abs*).
2. Given all program points, compute the entire space of possible MBTs (*MBT-rel*).

The latter enables the selection of appropriate preemption point candidates in the first place. Both problems are subject to this paper.

3.3 Fundamentals

We now define the relation of program structure to path problems which establishes a sound formal basis previously missing in [11]. . According to [15], every path in a control-flow graph $G = (V, E)$ can be interpreted as a string over E. For nodes $u, v \in V$, a *path expression* is a regular expression P *of type* (u, v) (written as P(u,v)), such that every string π in the language $L(P)$ is a path from u to v. The type of a path expression $P(u, v)$ can be recursively defined as follows:

i. If $P = P_1 \cup P_2$, then P_1 and P_2 are path expressions of type (u, v)

ii. If $P = P_1 \cdot P_2$, then there must be a unique vertex w such that P_1 is of type (u, w) and P_2 is of type (w, v)

iii. If $P = P_1^*$, then $u = v$ and P_1 is of type (u, u).

These rules define (i) alternative paths, (ii) concatenation and (iii) repetition, respectively. Path expressions describe all structurally possible but (yet) unbounded paths in a CFG.

The underlying algebraic structure of regular expressions is a Kleene algebra $\langle E, \cup, \emptyset, \cdot, \epsilon, ^* \rangle$ where $\cup$ is addition with neutral element $\emptyset$, $\cdot$ is multiplication with neutral element ϵ (the empty string), the additional operator *, and the order of operator precedence $^* > \cdot > \cup$.

We can derive a cost model over path expressions. Since P^* yields paths of unbounded length, *frequency expressions* [10] are paths expressions with Kleene star '$*$' replaced by an interval $[l, h]$ such that $P^{[l,h]}$ now denotes the finite expansion to $\{P^l \cup P^{l+1} \cup \cdots \cup P^h\}$ with $P^* = P^{[0,\infty]}$. Let $\omega(u) \in \mathbb{N}_0$ be the WCET of a basic block u. Then the function W denotes the costs of the longest path in $L(P(s,t)^{[l,h]})$ and is defined as:

$$W(P) = \begin{cases} -\infty & \text{if } P = \emptyset \\ 0 & \text{if } P = \epsilon \\ \omega(u) & \text{if } P = e = (u, v) \\ max(W(P_1), W(P_2)) & \text{if } P = P_1 \cup P_2 \\ W(P_1) + W(P_2) & \text{if } P = P_1 \cdot P_2 \\ W(P^L) + \cdots + W(P^H) & \text{if } P = P^{[l,h]} \end{cases}$$

In other words, the underlying algebraic structure of frequency expressions can be replaced by

$$\langle \mathbb{N}_0 \cup \{-\infty\}, max, -\infty, +, 0 \rangle$$

to obtain maximal path lengths. If $P = \emptyset$, then $-\infty$ denotes infeasibility. Note that the cost of an edge (u, v) is defined as the cost of node u.

We still need to define how path expressions are constructed. In a cyclic and reducible control-flow graph $G = (V, E)$ with forward edges $F \subseteq E$, let h denote a loop head. Then the path expression P_R from a source node s to a sink node t is computed recursively by:

$$P_R(s,t) = \begin{cases} \bigcup_{(u,t)\in F} P_R(s,u)(u,t) & \text{if } t \neq h \wedge s \neq t \\ (\bigcup_{(u,t)\in F} P_R(s,u)(u,t))P(t,t) & \text{if } t = h \wedge s \neq t \\ \epsilon & \text{if } s = t \end{cases}$$

$$P(h,h) = (\textstyle\bigcup_{(b,h)\in E\setminus F} P(h,b)(b,h))^*$$

A path expression $P_R(s,t)$ in the acyclic (first) case is the union of all paths leading to the predecessors u of node t and the edges leading to t. In the cyclic (second) case, if some node t is a loop head, then $P_R(s,t)$ is a prefix of all paths in the loop body ($P(h,h)$) from the head to its bottoms, back to its head.

We call paths from head to bottom ($P(h,b)$) the *kernels* of a loop. Every last iteration is an *exit path* and is represented by the expression $P(h,t)$ for a head h and some exit node t.

In irreducible graphs, not all paths pass through the head and therefore the first iteration need not be a kernel. Let I_v be the set of entries to a scope, then for all $i \in I_v$, $P(i,b)(b,h)$ is a prefix of all kernels, representing all first iterations. In general, first and last iterations need to be "peeled off" from loops in order to be exact in irreducible graphs.

The corresponding path expression $P(s,t)$ for those graphs (with $P(h,h)$ being identical to the equation above) is defined as:

$$P(s,t) = \begin{cases} \bigcup_{(u,t)\in F} P(s,u)(u,t) & \text{if } t \neq h \wedge s \neq t \\ (\bigcup_{(u,t)\in F} P(s,u)(u,t))P(t,t) \\ \quad \cup\, P'(s,h) & \text{if } t = h \wedge s \neq t \\ \epsilon & \text{if } s = t \end{cases}$$

$$P'(s,h) = \textstyle\bigcup_{i\in I_v\setminus\{h\}} P(s,i) \bigcup_{\substack{(b,h)\in E\setminus F \\ \wedge i \rightsquigarrow b}} P(i,b)(b,h)P(h,h)$$

Expressions from node s to node t, where t is a loop head, alternatively have a prefix $P(s,t)$, which denotes paths that enter the loop through its head, or prefixes $P(s,i)$ which denote paths to all other entries. In the first case, the first iteration coincides with kernels and is therefore modeled by $P(t,t)$. For all other entries, expressions P' are a composition of paths from entries to bottoms ($P(i,b)(b,h)$) and kernels ($P(h,b)(b,h)$), including the edges back to the head, respectively. Not all entries can reach all bottoms though: those paths are excluded. We call the set of paths $L(P(i,b))$ *entry paths*.

3.4 Path Constraints

The path model so far only reflects the structurally possible paths and the cost model which denotes its cost. In the following, we discuss the relation of flow bounds (c.f. Sec. 3.1) and bounds in frequency expressions.

For worst-case paths, just upper bounds are sufficient to compute WCEP. (Safe) Upper bounds are typically provided

by means of manual constraint annotation or value analysis [16]. Accuracy then depends on the expressiveness of constraints themselves. Flow bounds denote bounds on *node frequencies*[4] — the maximal number of occurrences of nodes on possible worst-case paths. Let the indicator function 1_e be defined as $1_e(u) = 1$ if $(u, \cdot) = e$, then the *multiplicity* m_π of a node u on a path π is defined as:

$$m_\pi(u) = \sum_{(u,\cdot)\in\pi} 1_{(u,\cdot)}(u)$$

Given a set of frequency constraints (flow bounds) $\mathcal{C}$, the subset of structurally possible paths $L(P, \mathcal{C}) \subseteq L(P)$ that satisfy the constraints is:

$$L(P, \mathcal{C}) = \{\pi \in L(P) \mid \forall C \in \mathcal{C} : C(m_\pi)\}$$

Constraints here are abstract. Later, in Sec. 3.5.3, we will define the concrete semantics of constraints for our specific model.

Explicit path analysis techniques typically trade accuracy for simplicity and performance by approximating constraints. The set $L(P, \mathcal{C})$ can be approximated by another expression P' such that $L(P, \mathcal{C}) \subseteq L(P')$ [10]. In practice, approximation is achieved by limiting constraints to backward edges (equivalent to loop bounds[5]), by overestimating frequencies of nested loops that depend on frequencies of enclosing loops (e.g. triangle loops) or by ignoring mutual exclusion of structurally possible but not realizable paths (e.g. `if x do a; if not x do b;`). IPET, so far, is the only exact approach that allows an unrestricted bounding of frequencies in all these cases. The constraint model we propose later solves the problem of bounding frequencies of individual nodes with annotation semantics to specifically model MBP.

The (abstract) constraints $\mathcal{C}$ restrict all repetitions relative to the current path expression P. If P specifies an entire program, then the constraints denote *global bounds*. Intuitively, however, loop bounds, for example, shall limit the repetition of paths for a single instance of a loop and thus denote *local bounds*. In particular, for the practical computation of WCEP, we need both kinds of bounds. Just local bounds as in [11] are not sufficient. Next, we first outline our basic algorithm. We then address global flow bounds specifically.

3.5 Basic Framework

In the following, we introduce a general algorithm for worst-case path computations and discuss our flow bound model. It efficiently solves the problem of computing maximal costs of paths in a CFG according to the model established in Sec. 3.3. We formally define the general problem of finding longest paths within individual scopes in Sec. 3.5.1 and propose a generalized flow bound model in Sec. 3.5.2, which serves as a basis for MBT analysis. In Sec. 3.5.3, we specialize this model and propose solutions to both problems *MBT-abs* and *MBT-rel* (c.f. Sec. 3.2). We generalize the approach in [11] by adding handling of global flow bounds.

The recursive definition of path expressions suggests a topological order in which nodes should be visited to instantly evaluate path expressions to obtain their costs. In acyclic graphs (such as loop bodies), this is indeed obvious. However,

[4]Derived from [10], which define edge frequencies to facilitate loop bounds.

[5]Edge frequencies are notoriously impractical for fine-grained constraints, which is why we define node frequencies.

$G = (V, E = F \cup B)$	Control flow graph
$\vec{G} = (V, E = F)$	Acyclic control flow graph
$\mathring{G} = (\mathring{V}, \mathring{E})$	Scope tree
Σ	Annotation labels
$\Pi \subseteq \mathcal{P}(\mathbb{N}_0 \times 2^\Sigma \times V(G))$	Path states
$\gamma : V(G) \mapsto \mathring{V}(\mathring{G})$	Scope membership
$\omega : V(G) \mapsto \mathbb{N}_0$	Node weight
$\delta : \Pi \mapsto \mathbb{N}_0$	Path length
$\sigma : \Pi \mapsto 2^\Sigma$	Path signature
$\theta : \Pi \mapsto 2^\Sigma$	Pending paths
$o : \Pi \mapsto \mathring{V}(\mathring{G})$	Path origin
$\alpha : V(G) \mapsto 2^\Sigma$	Node annotations
$\rho : \Sigma \mapsto \mathcal{P}(\mathring{V}(\mathring{G}))$	Annotation range
$\beta : \Sigma \mapsto \mathbb{N}_0$	Flow bounds

Table 1: Definitions

due to the cyclic recursion in the presence of loops, the computation of absolute path lengths requires more than a single pass over all nodes.

Recall that $\vec{G} = (V, F)$ denotes the DAG obtained from the CFG without backward edges and that $\mathring{G} = (\mathring{V}, \mathring{E})$ denotes the scope tree. Then *scope order* [11] is a topological order over $\vec{G}$ which guarantees that no node v outside a scope is visited prior to any inner node u with $(u, v) \in F$, before all nodes of that inner scope have been visited. The rationale is that general loops have multiple bottoms, exits are not necessarily bottoms, and we need to know the lengths of every path in the loop body first, to be able to compute (cyclic) path lengths from entries to all exits.

We concretize constraints and call them *annotations* α, such that for a node u the function $\alpha(u) \in 2^\Sigma$ denotes symbolic labels from the alphabet Σ. A flow bound β is a numeric value such that for a label $l \in \alpha(u)$ the bound $\beta(l) \in \mathbb{N}_0$ represents an upper bound on the frequency of node u. Since we need to support local as well as global bounds, we generalize the idea and define an *annotation range* ρ, where $\rho(l) \in \{\, par^+(\mathring{s}) \mid \mathring{s} \in \mathring{V}(\mathring{G})\}$ denotes the validity of an annotation with respect to scopes. A local annotation is one whose range is restricted to the current scope, a global annotation on the other hand remains valid in all scopes. For example, given a node u, a scope $\mathring{s}$, its parent scope $\mathring{t}$, $l \in \alpha(u)$, $n = \beta(l)$ and $[\mathring{s}, \mathring{t}] = \rho(l)$, then node u is bounded to at most n repetitions for all iterations of $\mathring{s}$ and $\mathring{t}$. Intuitively, the lower bound on the range of flow bounds is the innermost scope a bounded node is mapped to (i.e. $\gamma(u) = \mathring{s}$).

Under reducibility and loop bounds, for a single loop, all paths pass through the loop head and there only exists a single longest path in the loop body. Thus, for each iteration of an enclosing scope, a single path is a representative for all instances of the inner scope. To be exact in general CFG and under a general flow bound model, multiple instances as well as different iterations of the same scope need to be taken into account simultaneously. We represent every path $\pi \in L(P(s, t))$ by a *path state* $p = \ell(\pi)$, which is a tuple (δ, σ, o). For convenience, we define functions δ, σ and o over path states, such that $\delta(p) \in \mathbb{N}_0$ is a path length, $\sigma(p) \in 2^\Sigma$ is a set of annotation labels we refer to as *signature*, and $o(p)$ is the *origin* (source node) of a path. The signature of a path π, represented by path state p, is the set of annotation labels along π. Formally:

$$\sigma(p) = \alpha(o(p)) \cup \bigcup_{(\cdot,v)\in\pi} \alpha(v)$$

Algorithm 1 Outline of explicit path analysis

1 **let** main u =
2
3 **let** join u =
4 **let** local $Q = \{p \in Q \mid \gamma(u) = \gamma(o(p))\}$
5 **let** $Q = \max_\sigma(\bigcup_{(u',u)\in E(\vec{G})} P[u'])$
6 $I[u] \leftarrow Q \setminus \text{local } Q$
7 $P[u] \leftarrow \text{local } P \cup$ **if** $I[u] \neq \emptyset$ **then** $\{(0, \alpha(u), u)\}$ **else** $\emptyset$
8
9 **let** transfer u =
10 $P[u] \leftarrow \{(\delta(p) + \omega(p), \sigma(p) \cup \alpha(u), o(p)) \mid p \in P[u]\}$
11 **if** "all nodes of scope $\mathring{s}$ visited" **then**
12 **let** ex p q $v = \sigma(q) \cup \{l \in \sigma(p) \mid \text{par}(\mathring{s}) \in \rho(l) \wedge v \rightsquigarrow o(p)\}$
13 **let** re p $v = \{(\delta(q) + \delta(p), \text{ex } p \; q \; v, o(q)) \mid q \in I[v]\}$
14 **for each** w : exit $\mathring{s}$ **do**
15 $O[w] \leftarrow P[w]$
16 $P[w] \leftarrow \max_\sigma(\bigcup_{v\in\text{entry}(\mathring{s})} \bigcup_{p\in\text{unroll}(v,w)} \text{re } p \; v)$
17
18 **let** finish u =
19 **if** "u is head of scope $\mathring{s}$, not root" **then**
20 **for each** v : exit $\mathring{s}$ **do**
21 $P[v] \leftarrow O[v]$
22 **for each** v : entry $\mathring{s}$ **do**
23 **for each** w : entry parent $\mathring{s}$ **do**
24 $\text{offset}[v] \leftarrow \max\{\text{offset}[v], \text{"longest path } w \rightsquigarrow v\text{"}\}$
25 **for each** v : entry $\mathring{s}$ **do**
26 $\text{final}[u] \leftarrow \max\{\text{final}[u], \text{offset}[v] + \text{"longest path } v \rightsquigarrow u\text{"}\}$
27
28 $v \leftarrow$ "head of outermost scope u can reach itself"
29 **for each** w : "scope order from v" **do** transfer join w
30 **for each** w : "scope order from v" **do** finish u

A signature induces equivalence classes of path states $[p]$ through the relation $p \overset{\Sigma}{\sim} p'$ iff $\sigma(p) = \sigma(p')$ such that

$$[p] = \{\ell(\pi) \mid \pi \in L(P(s,t)) \wedge p \overset{\Sigma}{\sim} \ell(\pi)\}$$

for all paths from origin s to node t. For a set of path states Q, we can now define the function $\max_\sigma$ such that

$$\max_\sigma(Q) = \{p \mid [p] \in Q/\overset{\Sigma}{\sim} \wedge \forall q \in [p] : \delta(q) \leq \delta(p)\}$$

is the minimal set of path states of maximal length partitioned by signature, obtained from the quotient set $Q/\overset{\Sigma}{\sim}$.

After these definitions, we can now describe the principle outline of the algorithm. The intuition is that loops are successively virtually unrolled and inlined into enclosing scopes in a first pass. This yields maximal path lengths from entries and to exits of individual scopes. For the root scope, this yields an absolute path length from the start node. In other words, we compute one iteration of a scope including all iterations of subscopes, then compute all iterations of the scope itself. Thus, if only the longest path of entire tasks is of interest, we would already be done. A second pass exploits that besides longest paths to exits, we can now also compute longest paths from entries of parent scopes to entries of nested scopes to obtain absolute path lengths to all nodes. In other words, from the last iteration of the outermost scope, we compute the longest path to the last iterations of its nested scopes terminating in individual nodes.

To keep matters terse, we assume that all loops have just a single bottom, and entries and exits only connect directly adjacent scopes. Important definitions are summarized in Tab. 1. Any node can be a start node, which is handled as just an additional entry into all enclosing scopes [11]. We further assume that for every entry u to scope $\mathring{s}$, there implicitly exists an annotation $l \in \alpha(u)$ such that $\beta(l) = \infty$ denotes a flow bound of range $\rho(\sigma) = [\mathring{s}, \mathring{s}]$.

Alg. 1 is a naive version of the reference implementation proposed in [11], stripped of all optimizations and unnecessary details and extended to meet our requirements. We use an impurely functional pseudo language, where an expression $A \leftarrow v$ denotes assignment and *let* defines functions. As usual, function invocation is right-associative, so we can spare parenthesis. The arrays P, I and O hold path states. Initially they are empty. The arrays *offset* and *final* hold path lengths, initialized to 0. The function *unroll* performs *virtual unrolling* to compute longest paths from entries to exits and will be subject to detailed discussion in the following Sec. 3.5.1.

The computation is invoked with the start node u (l. 1) and starts at the *effective start node* v (l. 28) instead of the specified start node u. Starting in v guarantees that all paths required for virtual unrolling of all enclosing scopes will be correctly represented by path states. Two passes are then required to compute the longest paths.

In the first pass (l. 29), the longest path to the CFG exit is computed. The function *join* (l. 3) joins path states of predecessors. From a set of states, *local* (l. 4) extracts the ones of the same scope as node u, and Q (l. 5) is the minimal set of predecessor path states (max_σ). Then I (l. 6) holds the states of paths that just entered the scope and $P[u]$ (l. 7) is the set of "local" path states and a new state that represents the "entering" paths. The function *transfer* (l. 9) updates the set of path states (l. 10) by increasing the path length δ by the cost ω of the basic block and adding new annotations α to their signature σ, while keeping the path origin o. If all nodes in a scope are visited, then path states represent all simple paths from all entries (l. 11) and we can virtually unroll longest paths from all entries to all exits. The function *ex* (l. 12) extends the signature of state q by all labels in the signature of state p, whose range encompasses the parent of the current scope and whose origin can be reached from entry v, which effectively carries annotations over to a parent scope (non-local flow bounds in our case), but not beyond the point where the bounded node cannot be reached anymore. Beyond this point, the information is useless.

The function *re* (l. 13) restores path states of the parent scope which have previously been stored in the array I. Path state p represents a longest path from entry v to the current exit. The length δ is increased accordingly and annotations are extended by the non-local ones of state p. For each exit (l. 14), the original path states are stored in the array O (l. 15) before they are replaced in l. 16: here, the minimal set of longest paths that passed through all entries and which have now been updated by the longest paths through the scope is computed. The function *unroll* returns a set of path states representing the longest paths from entry to exit.

In a second pass (l. 30), function *finish* (l. 18) is invoked for each node. In l. 19, if node u is the head of a scope $\mathring{s}$ which is not the root scope, then we can obtain the longest path to all its entries by adding the maximal length to the entry of the parent, the maximal length from each parent entry to each own entry and the maximal length from this entry to the target node. In l. 21, the original states in all exits are restored, which have previously been overwritten (l. 16). The array *offset* (l. 24) holds maximal path lengths to each entry v, which is composed of the maximal length to

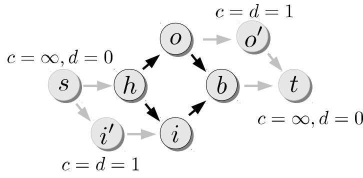


Figure 2: Flow network for virtual unrolling

Bound	Pivot	Min	Multiple	Remainder
$4^{[2,2]}, 6^{[1,2]}, 11^{[0,2]}$	2	$4^{[2,2]}$	$1^{[1,2]}, 2^{[0,2]}$	$2^{[1,2]}, 3^{[0,2]}$
$4^{[2,2]}, 2^{[1,2]}, 3^{[0,2]}$	1	$2^{[1,2]}$	$1^{[0,2]}$	$1^{[0,2]}$
$4^{[2,2]}, 2^{[1,2]}, 1^{[0,2]}$	0	$1^{[2,2]}$	–	–

Table 2: Instantiation of scopes with complex bounds

a parent entry w and a maximal length to v. Once all offsets are known, the maximal length to node u can be computed (l. 26). We deliberately keep l. 24, 26 vague, but it should be understood that longest paths are also obtained by means of the *unroll* function. The exact implementation depends on the flow bound model and the data representation.

3.5.1 Virtual Unrolling

Alg. 1 performs virtual unrolling of a scope to obtain longest paths from an entry to any node of a scope: to exits in the first pass and to all nodes in the second pass. This is a network flow problem. Specifically, we need to solve the maximal cost, maximal flow network flow with demand problem where costs, capacities, flows and demands are bound to nodes, not edges. Formally, the problem is defined as: Let $N = (V, E)$ be a weakly connected directed graph, with a source node s and a sink node t. With each node, a cost $w(u)$, a capacity $c(u)$, a flow $f(u)$ and a demand $d(u)$ is associated. The usual properties of flow networks and the additional invariant that $d(u) \leq f(u) \leq c(u)$ must hold. The problem is to maximize $\sum_{u \in V} f(u)w(u)$.

We can reduce the virtual unrolling problem as follows: For a scope, let i be an entry, o be any[6] node (of the same scope), h be its head and, without loss of generality, b be its bottom. Recall that in Alg. 1, a path state only represents path lengths (costs) and that subscopes are only accounted for by increasing this length once a subscope is finished. Therefore, let $G = (V, E)$ be an acyclic subgraph of the CFG such that all nodes V belong to the same scope, with its entry in h and its exit in b. Then let $N = (V, E)$ be the flow network where $V(N) = V(G) \cup \{s, t, i', o'\}$, $E(N) = \{E(G) \cup (s, i'), ('i, i), (o, o'), (o', t), (s, h), (b, t)\}$, $c(i') = d(i') = c(o') = d(o') = 1$, $c(s) = c(t) = \infty$ and $d(s) = d(t) = 0$. All other demands are equal to 0, and all capacities are equal to ∞, except where flow bounds are given. Fig. 2 depicts an example network, with the gray components being added to the original subgraph. Recall that entry, exit paths and kernel paths must be distinguished in irreducible CFG. The network reflects this requirement.

Solving this problem is easy since we already computed all paths in this network while traversing the nodes of a scope. Notably, we neither need to construct the network, nor do we need to know the paths explicitly: We know the length of all longest paths from all origins whose capacity is bounded by distinct sets of flow bounds. Let $s = \{l_0, \dots\}$ be the signature of such a path, then it cannot be repeated more often than $\min\{\beta(l_0), \dots)\}$ times. To account for the costs of a path to the overall objective value, we reduce the capacities of all nodes along this path by the minimum flow and repeat until no feasible path can be found anymore (reminiscent of the classic Ford-Fulkerson method). To maximize the overall cost, we first account for the entry and exit paths to satisfy the demand[7] (choose path states of respective origin), then sort kernels in descending order of length, accounting for them until all capacities are exceed.

Recall that paths are collectively and implicitly represented by path states. Every path that passes through the entry of a scope represents an iteration of a parent scope and induces a new instance which has to be virtually unrolled. Each virtually unrolled scope is represented by a path state, too. In Alg. 1, *unroll* returns these representatives which are subsequently *virtually inlined* by updating lengths and passing on annotations (l. 16). Unrolling has to be performed for all pairs of possible entry and exit paths [11].

3.5.2 General Flow Bounds

Virtual unrolling maximizes the path length and the flow with respect to the given flow bounds for a single instance of a scope. Such an instance serves as a representative for all instances due to iterations in the parent scope. Given just local bounds, each iteration of a parent scope that enters through a specific entry passes through exactly the same instance. In this case, we refer to an instance as being *context-free*. For example, local bounds model loop counters that are initialized when a loop is entered.

Bounds with ranges beyond the current scope make such instances *context-dependent*. Instances of a scope may be different for different iterations of a parent scope, despite entering through the same entry. For example, if loop counters are not initialized when entering the scope but only in a parent, then virtual unrolling depends on the iterations of this parent.

We assume a scope that models a loop consisting of just a single node (*self-loop*), and we write $\beta^{[\mathring{s}, \mathring{t}]}$ to denote a flow bound of value β and a range from an outer scope $\mathring{s}$ to an inner scope $\mathring{t}$. So, for example, $1^{[0,2]}$ denotes a flow bound such that the annotated node may only be traversed once in all instances of scopes 0 to 2, collectively. A bound of $1^{[2,2]}$ denotes that a node may be traversed once in every instance of the innermost scope 2 but is otherwise unbounded (context-free). Given the annotation $4^{[2,2]}, 6^{[1,2]}, 11^{[0,2]}$ as depicted in Tab. 2, which allows 4 iterations for each instance of the innermost scope 2, but only if the total number of iterations does not exceed 6, relative to scope 1, and does not exceed 11 for scope 0. To exactly model these bounds when virtually unrolling scope 2, we have to create multiple instances to reflect different iterations of scopes 1 and 0, respectively, such that flows are maximized. The rows in the table denote context-dependent instances of scope 2.

The minimal bound of the first instance is 4 (Min). Thus, in each iteration of the parent scope $\mathring{1}$, the loop node will be repeated 4 times. As a consequence, the parent iteration itself must additionally be bounded by $1^{[1,2]}, 2^{[0,2]}$ (Multiple),

[6]In the first pass of Alg. 1, node o denotes exit nodes only. In the second pass, it denotes all other interior nodes.

[7]Virtual unrolling is only feasible for loops that can be entered and left at all over feasible paths.

since repeating it more often would violate bounds of the self-loop ($4 \times 1^{[1,2]} \leq 6^{[1,2]}$, $4 \times 2^{[0,2]} \leq 11^{[0,2]}$). Obviously, bounds $6^{[1,2]}$ and $11^{[0,2]}$ have not been exceeded yet for this single instance of scope 2. For a different iteration of the parent scope, in another instance of scope 2, we would still be allowed to traverse according to the bounds $2^{[1,2]}, 3^{[0,2]}$ (Remainder), in addition to the local bound $4^{[2,2]}$. Local bounds must hold only for a single instance of a scope, so for each instance they are effectively reset. The resulting set of bounds that apply for the next instance of scope $\mathring{2}$ is depicted in the second row. With "Pivot" we denote the current reference scope "below" which bounds are reset, respectively. Thus, in the second instance we can traverse the node at most twice locally (Min), only once in the parent scope (Mul.) on the same path but yet once (Rem.) in another instance, for another iteration of scope 0. Note that context-dependent instances model flow bounds exactly and can be avoided by approximation [10].

In Alg. 1 l. 12, the function *ex* takes care of passing "Multiples" to the path states representing paths in parent scopes, thus effectively creating additional iterations explicitly represented by path states. In l. 13, each signature of a parent path state is updated accordingly. The first pass therefore outlines the logic for WCET under general flow bounds.

Computing context-dependent instances for general flow bounds in detail are subject to a general technical discussion of virtual unrolling, which is beyond the scope of this paper. In the following we will focus on a special case of flow bounds: For MBT, we need to take global flow bounds into account whose upper bounds equal 0. No iteration of any scope is allowed to pass through a specific node. This removes the demand for context-dependent instances

3.5.3 Maximum Blocking Times

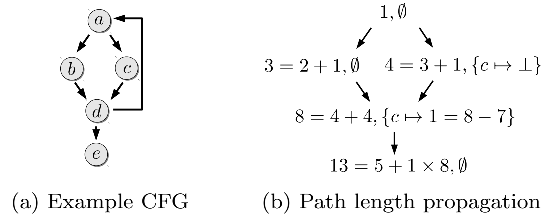
Maximum blocking times can be computed in two ways. For the sake of simplicity, we ignore context-dependence in the following.

Since in MBPs a sink node is allowed only as the terminal node, we can compute the MBT for a given set of preemption points by annotating each corresponding node with an additional global flow bound of value 0. Starting the analysis from each of these nodes then yields the MBP to all other reachable points. Effectively, adding such global bounds cuts the CFG in disjoint subgraphs, in each of which we compute the WCET: No path can traverse a preemption point. The idea corresponds to the IPET formulation in [3] and Alg. 1 outlines the implementation of an explicit path analysis for this. This solves *MBT-abs* as stated in Sec. 3.2.

In a naive implementation, it would be sufficient to compute the MBT for each pair of source and sink nodes. This, however, is extremely inefficient as redundancy between these distinct problem instances is significant. Ideally, we would like to reuse Alg. 1 to compute MBT from a start node to all reachable nodes.

The central problem in achieving this is that every node is a potential sink, while it is also just a node on paths to other sinks. Defining global bounds as proposed above is impossible. Instead, bounds must be applied conditionally, relative to each sink.

The intuition for our solution is that a path is a potential MBP for all nodes that have not been reached yet. Inversely, it can not be a potential MBP for all nodes already on this path. In terms of virtual unrolling and inlining, any



(a) Example CFG (b) Path length propagation

Figure 3: Length computations with difference encoding

path becomes infeasible for all nodes already traversed and therefore must not be used to compose longer paths to these nodes. Conceptually, flow bounds still include a global bound for sink nodes. Technically, however, we now distinguish between bounds that model actual program semantics and those that bound sink nodes.

By definition, the MBP to the CFG exit node corresponds to its WCEP. It is the only path to a sink node that is guaranteed to only contain the sink once. And it is entirely unaffected by relative path infeasibility. We refer to the WCEP to each program point as a respective *reference path*.

In general, WCEP and MBP to any sink node share a common prefix. Differences in the suffix of a path result from relative infeasibility only (all other flow bounds apply unconditionally) and can only occur within the same strongly-connected component (c.f. Fig. 1(a)).

We can exploit these insights in Alg. 1 by only making minimal changes to the *transfer* function and changes in the representation of path lengths. Informally, in each transfer step, the current node is marked infeasible for all path states in this node, so that they will never be used to compose an MBP to it. This implies that all MBP are represented by the reference path for all nodes that are not already on it. Once a node is on the reference path, its MBP will have to be maintained separately. For convenience, we store difference in length from the reference path only and designate a special difference value "$\perp$" to denote relative infeasibility.

Fig. 3 illustrates the proposed difference encoding. For the CFG to the left, we assume node costs 1 to 5 for nodes a to e, a loop bound of 1 in the bottom node d. We reduce the illustration to just the reference path and the MBP to node c. The tuples to the right denote the proposed length encoding. Syntactically, in (l, D), l denotes the length of the reference path, and D encodes deviations from it. In nodes a and b, both paths coincide. In node c, the path is marked infeasible (denoted by $c \mapsto \perp$) for any MBP towards node c. In node d, since the path through node b is the longest path, we update the difference information on c: Not all paths are infeasible regarding node c. We account for this by encoding the difference from the reference path. We can now virtually unroll the loop separately for the reference path and for node c. This would already yield the correct MBT to c $((abdac))$. In node e, information on node c can be discarded, since it is not reachable anymore, so no longer MBP to node c could be composed.

Recall the cost model in Sec. 3.3 which denotes an algebra with operators max and + over $\mathbb{N}_0$. According to this, Alg. 1 computes worst-case paths. Now, instead of natural numbers to represent a single path length, we define $\mathbb{L} = \mathbb{N}_0 \cup \{\perp\}$, where $\perp$ denotes infeasibility, and a new algebraic structure $\langle \mathbb{L}, max_{\mathbb{L}}, \perp, +_{\mathbb{L}}, 0 \rangle$ where $+_{\mathbb{L}}$ with identity element 0 is

defined as

$$l_i +_{\mathbb{L}} l_j = \begin{cases} l_i + l_j & \text{iff } l_i \neq \bot \wedge l_j \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

And where $max_{\mathbb{L}}$ with identity element $\bot$ is defined as:

$$max_{\mathbb{L}}(l_i, l_j) = \begin{cases} max(l_i, l_j) & \text{iff } l_i \neq \bot \wedge l_j \neq \bot \\ l_i & \text{iff } l_i \neq \bot \wedge l_j = \bot \\ l_j & \text{iff } l_i = \bot \wedge l_j \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

A path state then encodes the lengths for all MBP simultaneously, if we replace $\mathbb{N}_0$ to represent length in a path state with elements from $\mathbb{D} \subseteq \mathcal{P}(\mathbb{N}_0 \times (\mathcal{P}(V(G) \times \mathbb{L}))$, whose algebra is defined by $\langle \mathbb{D}, max_D, 1_{max}, +_D, 1_+ \rangle$ where $\mathbb{D}$ represents the length of a reference path and the differences to the length of the reference path. Identity elements and operators are derivatives of the algebra on $\mathbb{L}$. Addition with identity element $1_+ = (0, \{u \mapsto 0, \dots\})$ is defined as:

$$(r_a, \{u \mapsto l_i, \dots\}) +_D (r_b, \{u \mapsto l_j, \dots\}) = \\ (r_a + r_b, \{u \mapsto l_i +_L l_j, \dots\})$$

Maximum with identity element $1_{max} = (\bot, \{u \mapsto \bot, \dots\})$ is defined as:

$$max_D((r_A, \{u \mapsto l_i, \dots\}), (r_b, \{u \mapsto l_j, \dots\})) = \\ (max(r_a, r_b) = r, \{u \mapsto (r - max_L(r_a - l_i, r_b - l_j)), \dots\})$$

Differences only need to be maintained as long as its respective sink node is reachable. Note that this representation does not lose information and explicit path states for each individual node can always be restored. This is indeed necessary for virtual unrolling which now has to be performed for each explicitly recognized sink separately, since sets of feasible paths are potentially individual. With this encoding, we compute the MBP for all reachable nodes with only minimally changes to the base algorithm. This solves *MBT-rel* as stated in Sec. 3.2.

Computing WCET or MBT from and to all nodes requires a run for each source node. However, paths and unroll results can often be reused between runs from different sources, reducing the overhead. It is also important to note that virtual unrolling does not have to be performed as often as suggested in Alg. 1, but only for flow bounded nodes. Path lengths to individual nodes can be easily derived from lengths to bounded nodes on the same path [11].

4. EVALUATION

We evaluate the average performance of our MBT analysis in the following. The aim is to demonstrate its scalability characteristics for typical control flow graphs of varying sizes and topologies. We perform runtime measurements on the Mälardalen WCET benchmark suite (MRTC [1]) as well as on control-flow graphs generated from random syntax trees (AST). For these graphs, sizes vary between 10 and approximately 12,000 CFG nodes. An AST is composed of four high-level language constructs IF, IFELSE, WHILE and DOWHILE. Entries, exits as well as node costs and flow bounds are generated in addition. Probabilities of components can be specified, such that specific use-cases can be isolated.

We compare results for *MBT-rel* against *IPET* [13] and the WCET analysis proposed in [11], we refer to as *PAAN* in the following. Without loss of generality, the latter two effectively compute *MBT-abs* for just two preemption points

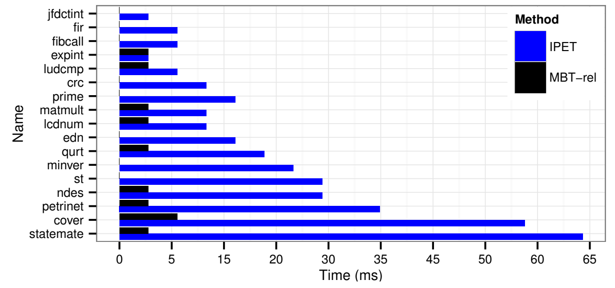


Figure 4: Runtimes on real-time benchmarks

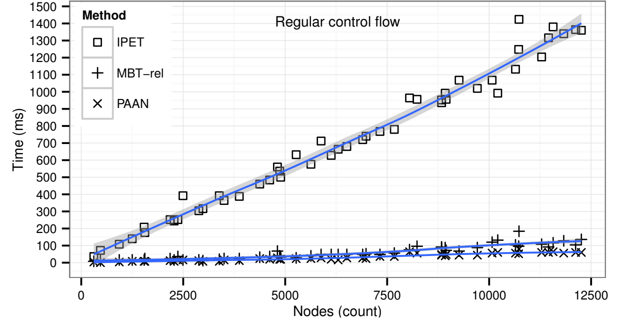


Figure 5: Runtimes on non-degenerated graphs

that coincide with the CFG entry and exit, respectively. The rationale for the restriction to just two dedicated preemption points is the limitation of IPET: The IPET-based MBT analysis proposed in [3] is precisely a WCET IPET model, if no other preemption points but a task's entry and exit are specified. A larger set of preemption points cuts the graph into subgraphs, such that the ILP objective value only yields the length of the longest WCEP among these subgraphs. It is not possible to obtain the results for each subgraph separately unless a single ILP model per subgraph is generated, which then would yield just another problem for two dedicated program points of a single CFG. Note that *IPET* does not yield any result for intermediate nodes. *PAAN*, in addition, computes the WCET to all nodes and *MBT-rel* computes the MBT to all nodes from the entry node, respectively.

The experiments are carried out on a single core of an *Intel Xeon E5630* (2.53GHz) CPU. We measure the accumulated CPU time of all computing steps for *MBT-rel* and *PAAN*, including all preprocessing (e.g. scope tree construction). For *IPET*, the generation of linear equations is included. The IPET ILP is solved using *CPLEX* (v. 12.4) with default arguments. All analyses are carried out in succession on the very same CFG.

Fig. 4 shows the results for a random subset of MRTC benchmarks at a resolution of 1ms. *MBT-rel* significantly outperforms *IPET* in all use cases. We solve the MBT problem from the source to all reachable nodes below 1ms in some cases. These hard real-time benchmarks are comparably small in size (ca. 50 to 1200 LOC). Thus, scalability for very large problem instances is not obvious. Large non-real time benchmarks lack the mandatory annotations. Therefore, we chose to synthesize benchmarks.

Fig. 5 depicts the average performance for CFGs that we consider "typical". 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 addi-

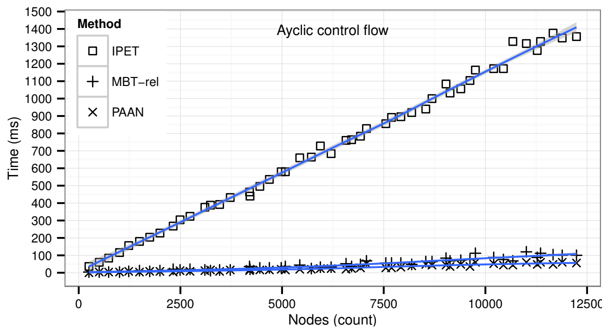


Figure 6: Runtimes on acyclic graphs

tional flow bounds of 0.1, a maximal loop depth of 3 and a single flow bound per loop. The chart illustrates the significant performance advantage of both *MBT-rel* and *PAAN* over the ILP solution. We improve over *IPET* by about a factor of 10, while computing MBTs to all sink nodes at once. The scalability is excellent in all three approaches (practically linear). This shows that extremely large problem instances can be practically solved quickly. All three yield identical results here (except for the fact that *PAAN* and *MBT-rel* solve the problem for all nodes, *IPET* does not — it would require n runs or a very large ILP model). Even though computations and the state space are potentially more complex, the difference of *PAAN* to *MBT-rel* is practically negligible.

Fig. 6 illustrates results for acyclic graphs. Probabilities are proportional to the standard case above, with the exception of P(WHILE) = P(DOWHILE) = 0. The lack of loops means that the only MBPs are direct paths to each node. No virtual unrolling for individual nodes is ever performed. Thus, the chart depicts the baseline for all MBT analyses. As can be seen from this, loops have practically no effect.

The scalability of *PAAN* and *MBT-rel* is highly correlated and substantiate our claim that, either in the WCET or in the MBT case, *IPET* is not competitive. In [11], the authors have thoroughly studied *PAAN* for other "degenerated" (irreducible) graph topologies. In those cases, we found scalability characteristics of *PAAN* compared to *MBT-rel* similar to those presented here. The primary reason for the "resiliency" of *MBT-rel* and *PAAN* against complex graph topologies is that the state space grows exponentially only with the complexity of the annotations, not with the graph size. Since no algorithmic recursion is involved, the number of loops and the complexity of their nesting are practically irrelevant. Note that in these experiments, *MBT-rel*, *PAAN* and *IPET* yield identical path lengths, where comparable.

5. CONCLUSION AND FUTURE WORK

We proposed a general, accurate and fast analysis technique for the computation of maximum blocking times. Its framework significantly improves upon the state of the art in explicit path analysis by introducing a general flow bound model for exact explicit analyses and a specialization to global infeasibility bounds that allows for the efficient computation of MBT. We showed that IPET can be outperformed significantly, even for problems with global flow bounds like MBT, despite computing paths from a single source to all sink nodes at once, not to a specific one. To this end, we proposed an efficient compression for path representations. Specifically, for deferred preemption schedules, this now allows the practical exploration of design spaces in terms of MBT and explicit preemption points.

Our approach dominates alternative approaches in terms of generality and accuracy. MBT now allow for precise preemption point placement in general CFG. To completely replace IPET in practice for MBT as well as for WCET analysis, an efficient way to model mutual exclusion of paths is required, and generalized flow bounds must be further discussed in all detail. We will address these limitations specifically and propose an analysis framework that has the potential to yield significant gains in performance and accuracy for static multitask timing analysis. A generally open problem is modeling of dependent loop counters (e.g. triangle loops) which can be solved by supporting lazy evaluation by means of symbolic path expressions.

6. REFERENCES

[1] Mälardalen WCET benchmark suite. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

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

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

[4] S. Altmeyer, R. I. Davis, and C. Maiza. Improved Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems. *RTS*, 48(5):499–526, 2012.

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

[6] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. C. Buttazzo. Optimal Selection of Preemption Points to Minimize Preemption Overhead. In *Proc. of ECRTS*, 2011.

[7] G. C. Buttazzo, M. Bertogna, and G. Yao. Limited Preemptive Scheduling for Real-Time Systems. A Survey. *IEEE TII*, 9(1):3–15, 2013.

[8] S. Bygde, A. Ermedahl, and B. Lisper. An Efficient Algorithm for Parametric WCET Calculation. In *RTCSA*, pages 13–21. IEEE Computer Society, 2009.

[9] L. Carter, J. Ferrante, and C. D. Thomborson. Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger. In A. Aiken and G. Morrisett, editors, *POPL*, pages 106–114. ACM, 2003. ACM SIGPLAN Notices 38(1), January 2003.

[10] B. Huber, D. Prokesch, and P. Puschner. A Formal Framework for Precise Parametric WCET Formulas. In T. Vardanega, editor, *WCET*, volume 23 of *OASIcs*, pages 91–102, 2012.

[11] J. Kleinsorge, H. Falk, and P. Marwedel. Simple Analysis of Partial Worst-case Execution Paths on General Control Flow Graphs. In *Proc. of EMSOFT*, 2013.

[12] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling. *IEEE TC*, 47(6):700–713, June 1998.

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

[14] F. Stappert, A. Ermedahl, and J. Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proc. of CASES*, pages 132–140, 2001.

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

[16] 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.