Precise and efficient parametric path analysis

Ernst Althaus, Sebastian Altmeyer, Rouven Naujoks

Angaben zur Veröffentlichung / Publication details:

Althaus, Ernst, Sebastian Altmeyer, and Rouven Naujoks. 2011. "Precise and efficient parametric path analysis." In *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems - LCTES '11, April 2011, Chicago, IL, USA*, edited by Jan Vitek and Bjorn De Sutter, 141–50. New York, NY: ACM Press. https://doi.org/10.1145/1967677.1967697.

Precise and Efficient Parametric Path Analysis

Ernst Althaus
Johannes-Gutenberg-Universität Mainz
ernst.althaus@uni-mainz.de

Sebastian Altmeyer
Universität des Saarlandes
altmeyer@cs.uni-saarland.de

Rouven Naujoks
Max-Planck-Institut für Informatik
naujoks@mpi-inf.mpg.de

Abstract

Hard real-time systems require tasks to finish in time. To guarantee the timeliness of such a system, static timing analyses derive upper bounds on the worst-case execution time (WCET) of tasks. There are two types of timing analyses: numeric and parametric. A numeric analysis derives a numeric timing bound and, to this end, assumes all information such as loop bounds to be given a priori. If these bounds are unknown during analysis time, a parametric analysis can compute a timing formula parametric in these variables. A performance bottleneck of timing analyses, numeric and especially parametric, is the so-called path analysis, which determines the path in the analyzed task with the longest execution time bound. In this paper, we present a new approach to path analysis. This approach exploits the often rather regular structure of software for hard real-time and safety-critical systems. As we show in the evaluation of this paper, we strongly improve upon former techniques in terms of precision and runtime in the parametric case. Even in the numeric case, the approach competes with state-of-the-art techniques and may be an alternative to commercial tools employed for path analysis.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods

General Terms Performance, Verification

1. Introduction

Hard real-time systems require tasks to finish in time. To guarantee the timeliness of such a system, static timing analyses derive upper bounds on the worst-case execution time (WCET) of tasks. To be useful in practice, timing analyses must be

- *sound*, to ensure the reliability of the guarantees,
- *precise*, to increase the chance to prove the satisfaction of the timing requirements, and
- *efficient*, to make them useful in industrial practice.

The high complexity of modern processors and modern embedded software hampers analyses to achieve all three properties at once. Exhaustive measurement, for instance, may be sound and precise but is infeasible for realistically sized programs. Simple end-to-end measurements are easy to derive, but are possibly unsound. Static timing analyses may derive sound upper bounds on the execution time by construction. In general, the analyzed programs are represented as control-flow graphs (CFG) with basic code blocks as nodes and edges representing the possible execution paths. A set of static analyses compute—amongst other information—upper bounds on the execution time of the basic blocks and upper bounds on the number of loop iterations. The execution time bound of the task is then given by a path P in the CFG for which the sum of occurrences of basic blocks of P times their execution time bound is maximal. The step of the timing analysis that computes this path is usually referred to as *path analysis*.

Timing analysis handling only numeric values as bounds on the maximal number of loop iterations are referred to as numeric timing analyses. The drawback of these analyses is that information such as bounds on the maximal number of loop iterations must be known statically, i.e. during design time. Some systems need guarantees for timely reactions which are not absolute, but dependent on a numerical parameter. In such cases, numeric timing analyses offer only two possibilities. Either one provides bounds for the unknown variables or starts a new analysis each time the task is used with different values. The first option endangers precision, the second may unacceptably increase the analysis time. Parametric timing analyses circumvent this problem. Instead of computing numeric bounds valid for specific variable assignments only, parametric analyses derive symbolic formulas representing upper bounds on the task's execution times.

A performance bottleneck of the numeric and especially of the parametric timing analysis, is the path analysis. In the parametric case, also the precision of the execution time bound is limited by this step, as we will explain later. State-of-the-art techniques formulate the computation of the timing bound as a search for the longest valid path in the control-flow graph. A path is considered valid, if it respects bounds on the number of times a loop can be traversed for each time, the loop is entered.

In this paper, we propose a new approach to path analysis, both for numeric and parametric timing guarantees. Embedded software often exhibits a very regular structure. Based on this observation, the new algorithm exploits this regularity to improve on the performance in both cases and on the precision in the parametric case.

The paper is structured as follows: We present a general timing analysis framework and related work regarding path analysis in Section 2. In Section 3 we explain our new method, provide a correctness proof and argue about the performance. Evaluation of the method then follows in Section 5. Section 6 concludes the paper.

2. Timing Analysis & Related Work

In this section, we explain the different steps of the timing analysis focusing on the most eminent parts with respect to our new approach. The related work, to which we compare our method, is also given in this section.

2.1 Timing Analysis

Static timing analyses represent programs to be analyzed as *control-flow graphs* (CFG) $G = (V, E)$. A sequential list of instructions with a unique entry and a unique exit point are so-called basic blocks which constitute the nodes of the CFG. The edges of the CFG represent the possible control flow between the basic blocks. For the sake of simplicity, we can assume a single, unique entry and even a single, unique exit node of the CFG.

The first step of the timing analysis, the *CFG reconstruction* generates this CFG from the executable (see Figure 1 for the toolchain). Note that timing analysis has to resort to the level of the executable. Source code analysis can only deliver rough and possibly unsound estimates.

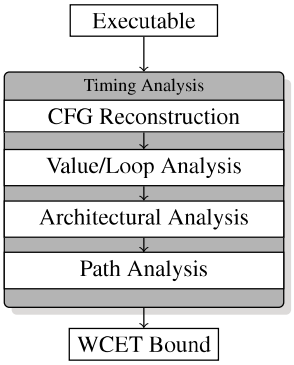

Figure 1. Timing Analysis Toolchain

To be able to derive an upper bound on the execution time of the analyzed tasks, the timing analysis has to compute upper bounds on the execution times (architectural analysis) for each basic block and upper bounds on the number of iterations of each reachable loop within the CFG (value/loop analysis). In case of a numeric analysis, the loop bounds are given as numeric values ($\in \mathbb{N}$); in case of a parametric analysis, loop bounds are represented by variables. Our new approach adapts the last step of a static timing analysis, the path analysis. Input to this step is only the CFG, the loop bounds, and the execution time bounds for each basic block, which we assume to be given.

Further information about static timing analysis can be found in [10].

2.2 Numeric Path Analysis

Path analysis combines the timing information for each basic block and the loop bounds and searches for the longest path within the executable. In this fashion, it computes an upper bound on a task's execution time. Searching the longest path is done using a technique called *implicit path enumeration* (IPET [11, 14]): the control flow graph and the loop bounds are transformed into *flow constraints*. The upper bounds for the execution times of the basic blocks as computed in the cache and pipeline analysis are used as weights. Figure 2 provides an example. The variables n_i, also called traversal counts, denote how often a specific edge is traversed. The first and the last basic block are left, resp. entered, exactly once ($n_1 = 1$; and $n_3 + n_6 = 1$;). For all other basic blocks, the sum of the traversal counts entering equals the sum leaving. The loop body (basic block 4, bounded by b_{loop}) is executed at most b_{loop} times as often as the loop is entered ($n_4 <= b_{loop} n_2$;). The constant c_j denotes the cost of the basic block j. The maximum sum over the costs of a basic block times traversal counts entering it determines the final WCET bound.

The resulting ILP can be solved by any solver. In practice, CPLEX is often used as a commercial and lp_solve as a non-commercial solution. In general, solving ILPs takes exponential time. In practice, however, ILP solver often exhibit a much better performance. In the evaluation, we present runtime of lp_solve and CPLEX and compare them to our new approach. The main focus of this work, however is on the parametric path analysis.

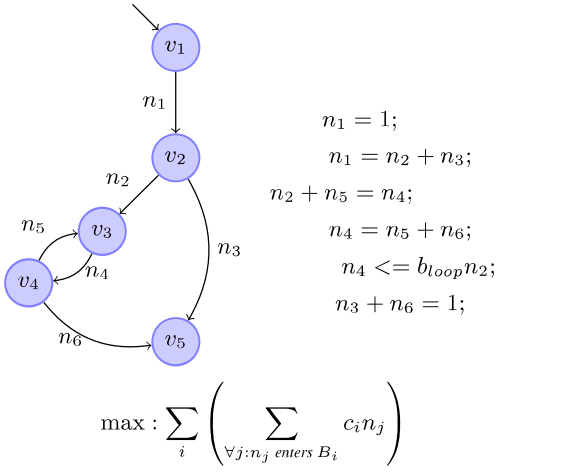

Figure 2. Control flow graph and the corresponding flow constraints

An alternative to implicit path enumeration is the tree-based approach by Puschner and Koza [13]. Formulas for specific control constructs, such as if-then-else or for-loops, are used to derive a timing formula for the complete syntax tree, which then delivers the timing bound. So, in contrast to IPET and our new approach, tree-based timing analyses resort to the source-code level and require special treatment of each program construct.

2.3 Parametric Path Analysis

Parametric path analysis computes the WCET formula by symbolically searching the longest execution path in the program. As in numeric timing analysis, implicit path enumeration is used to generate flow constraints. These flow constraints, however, must be linear in order to be used in an ILP. The only difference with respect to numeric analysis is the type of the loop bounds. Regarding the example in Figure 2, numeric analysis considers b_{loop} to be a numeric value and thus, computes a numeric WCET. The parametric analysis has to compute a WCET formula in the parameter b_{loop}. Although this seems to be a rather small difference, it has severe consequences. The expression

$$n_4 <= b_{loop} n_2;$$

is a linear constraint, if b_{loop} is a numeric value. However, the same constraint is non-linear in case of a parametric loop bound. Such non-linear constraints are caused by relative loop-bounds. As a solution, all relative loop bounds have to be converted to absolute ones. In the example in Figure 2, the variable n_2 is bounded by 1. Thus, the relative loop bound can be replaced by

$$n_4 <= 1 * b_{loop};$$

Such a replacement is possible in general. To replace loop bounds, each variable must be bounded. If we assume a variable to be unbounded, the whole ILP would be unbounded, too. Note that there are only non-negative execution time bounds for basic blocks. Hence, we can disregard such cases and can assume each variable to be bounded. In case of nested parametric loop bounds, however, a variable may be bounded by another parametric loop bound

only. Assume $n_o <= b_o$ to be the absolute loop bound of an outer loop, and $n_i <= b_i n_o$ the relative of the inner loop. After conversion, the absolute loop bound of the inner loop is $n_i <= b_i b_o$, which contains a non-linear term. The solution to this problem is to replace $b_i b_o$ by a new symbolic loop b' and to use $n_i <= b'$ as the loop bound constraint for the inner loop. Note that this step increases the number of parameters, and hence, the complexity of the symbolic ILP. In addition, absolute loop bounds lead to a lower precision than relative ones. Relative bounds still respect the relation to the loop entry edges. In case a loop is not part of the longest execution path through the CFG, a relative loop bound will not increase the execution time bound. In contrast, absolute loop bounds will contribute to the upper bound no matter if the corresponding loop is part of the worst-case path or not (see 5 for the details).

The constraint system after the conversion is completely linear and thus forms a valid symbolic integer linear problem. This ILP is then solved by a symbolic ILP-solver as proposed by Feautrier [8] using symbolic versions of the simplex [7] and cutting plane algorithm [9]. A free symbolic solver called *PIP* is also available. Note that the precision of the parametric path analysis via PIP and especially the runtime is rather poor. As we show in the evaluation later on, only small benchmarks can be solved using PIP.

2.4 Other approaches to Parametric Timing Analysis

Lisper proposed a parametric timing analysis [12] which has recently been implemented and extended by Bygde [4]. In contrast to our approach, they use a polyhedral abstract interpretation to compute the loop bounds and to create an ILP for the path analysis at once. Hence, the constraints used within their approach differ from the standard IPET model. In addition, their analysis—or at least the implementation of it—resorts to the level of the source code. Thus, a direct comparison between both approaches is not possible. As also observed in [2], the parametric ILP was the bottleneck and has been replaced by a method called *minimum propagation analysis* [4] based on a tree-like representation of the parametric formula. The better performance of the new approach comes at the cost of a lower precision which is often less than one percent but may reach up to 30% for some benchmarks. The paper of Bygde et al. [4] provides detailed experimental results, but is missing a theoretical performance analysis.

Other approaches to parametric timing analysis [3, 5, 6, 15] have a completely different structure and resort to the source-code level. Hence, the new path analysis technique, which we explain in the next section, can not be applied to these approaches.

3. Longest Paths in Singleton-Loop-Graphs

Embedded software for safety-critical systems is written with verification in mind or is even automatically generated. In both cases, program code often exhibits a very regular structure. As a consequence, control-flow graphs of embedded tasks are often reducible graphs that contain natural loops only. Or, at least, most loops in embedded software are reducible. In our new method, we exploit one special property of such natural loops, namely the fact that each such loop has a unique entry node. To emphasize this property, we refer to such loops as *singleton loops* and to graphs that contain only singleton loops as *singleton graphs*.

Exploiting the special structure of singleton graphs, we will show how to compute longest execution paths with high efficiency. The proposed algorithm is able not only to cope with numeric loop bounds (in polynomial time), but also to handle symbolic loop bounds (in output-polynomial time).

The algorithm itself is only applicable to singleton graphs. However, if non-singleton loops occur, we can still transform any control-flow graph into a singleton graph.

In this section, we first provide basic definitions, followed by an explanation of the algorithm. Then, we analyze its timing behavior, give a correctness proof and show how to transform arbitrary control-flow graphs into singleton graphs.

Note that this paper focuses on the practical aspects of the new path analysis. Further theoretical results can be found in [1].

3.1 Preliminaries

Before we describe the algorithm, it is necessary to formally define what we mean when talking about a loop in a control flow graph.

Definition 1 (loop). *Given a directed graph $G(V, E)$, we call a strongly connected component $S = (V_S, E_S)$ in G with $|E_S| > 0$, a* loop *of G.*

We denote by $\text{loops}(G)$, the set of all loops of G. We demand E_S to be nonempty. Otherwise, each isolated node would also be considered a loop.

Definition 2 (entry node). *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$ of G, we call $\mathscr{E} \in V_L$ such that there exists an edge $(u, \mathscr{E})$ in the cut-set*

$$\delta_G^+(V \setminus V_L) := \{(v', v) \in E \mid v' \in V \setminus V_L, v \in V_L\}$$

an entry node *of G.*

Given the control-flow graph depicted in Figure 2, nodes v_3 and v_4 constitute one loop. The entry node of this loop is node v_3.

Our algorithm relies on a special property of well-structured loops. Hence, we only consider loops exhibiting this property, which we define as follows:

Definition 3 (singleton loop). *A loop L in a graph G is called a* singleton loop *if L has exactly one entry node e.*

For the unique entry node of a singleton loop L, we write $\mathscr{E}(L)$. Note that we show in Section 3.6 how to handle loops of arbitrary structure.

The next two definitions lift the single-entry property from loops to graphs.

Definition 4 (sub-loops). *Given a loop $L = (V_L, E_L)$, we define*

$$\text{sloops}(L) := \bigcup_{v \text{ is entry node of } V_L} \text{loops}(G_v)$$

where G_v is the subgraph induced by $V \setminus \{v\}$.

Definition 5 (induced sub-loops). *Given a loop $L = (V, E)$, we call the recursively defined set*

$$\text{iloops}(L) := \{L\} \cup \left(\bigcup_{L_s \in \text{sloops}(L)} \text{iloops}(L_s) \right)$$

the set of induced sub-loops *of L.*

For a graph G, we extend the definition of iloops to graphs:

$$\text{iloops}(G) := \bigcup_{L_s \in \text{loops}(G)} \text{iloops}(L_s)$$

We call a graph G a *singleton-loop graph* if each induced sub-loop of G is a singleton loop. For such a graph, we write $\mathscr{E}(G) := \{\mathscr{E}(L) \mid L \in \text{iloops}(G)\}$ to denote the set of entry nodes of all induced sub-loops in G.

This far, we only argued about entry nodes of loops. Now, we formally define nodes and edges leaving a loop.

Definition 6 (portal nodes, transit edges). *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$ in G, we call*

$$\mathcal{T}(L) := \delta_G^+(V_L)$$

the set of transit edges *of L, i.e. the edges leaving the loop L, and*

$$\mathcal{P}(L) := \{p \in V_L \mid \exists\, (p, v) \in \mathcal{T}(L)\}$$

the set of portal nodes *of L, i.e. the last nodes over which a path can leave the loop L.*

In the control-flow graph depicted in Figure 2, node v_4 is a portal node and edge (v_4, v_5) a transit edge. Note that there is a one-to-one correspondence between singleton loops and their entry nodes, which justifies the following definition.

Definition 7 (loop-bound function)**.** *Given a singleton-loop graph $G = (V, E)$, we call a function*

$$\mathrm{b}\colon \mathscr{E}(G) \to \mathbb{N} \cup \{+\infty\}$$

a loop-bound function *for G.*

The loop-bound function is provided by the value/loop analysis, which is part of the timing analysis framework (see Section 2.1). We assume that all loops are bounded. If there is at least one unbounded loop, the worst-case execution time can not be computed. In parametric timing analyses, loops may be bounded by parameters, too. In such a case, the final timing bound is parametric in these parameters.

Now, we have to classify the valid paths, i.e. the paths that respect the loop bound conditions. If for a loop L, a loop bound of $\mathrm{b}(\mathscr{E}(L))$ is given, we mean that an execution path is not allowed to enter L and iterate on L more than $\mathrm{b}(\mathscr{E}(L))$ times, before the path leaves L again.

Definition 8 (valid path)**.** *Given a singleton-loop graph $G = (V, E)$, two nodes $s, t \in V$ and a loop-bound function b for G, we call a path $P := s \rightsquigarrow t$ a* valid path *if for all $L := (V_L, E_L) \in \mathrm{iloops}(G)$ and for all sub-paths $(\mathscr{E}(L), v_0, v_1, \ldots, v_k)$ of P with $v_i \in V_L$, the sub-path $(\mathscr{E}(L), v_0, v_1, \ldots, v_{k-1})$ contains at most $\mathrm{b}(\mathscr{E}(L))$ times the node $\mathscr{E}(L)$.*

The problem that we consider in the following is to determine the longest valid paths from a single source node s to all other nodes in G with respect to a given edge weight function. At first glance, this might seem more complicated than finding only the longest path from a single source node to a single destination node, but as in the computation of shortest paths in graphs, these problems are equally hard.

In the following, we will write $\mathrm{lps}(G, s, t)$ for a longest valid path from a node s to a node t and $\overline{\mathrm{lps}}(G, s, t)$ to denote the longest valid path from s to t that contains t exactly once. Most of the times, we will limit the discussion to the task of computing just the path weights for sake of simplicity. Note that this is not a real limitation, since the algorithm can easily be extended to also cope with the problem of reporting the paths as well.

In the following, we will assume that for each $v \in V$ there is a path from s to t containing v. All other nodes can be removed by a preprocessing in time $O(|V| + |E|)$. The resulting graph has at least $T - 1$ edges.

3.2 The Algorithm

Let us recall that a problem instance is given by a singleton-loop graph $G = (V, E)$, a source node $s \in V$, an edge weight function $w\colon E \mapsto \mathbb{N}$ and a loop-bound function $\mathrm{b}\colon \mathscr{E}(G) \to \mathbb{N} \cup \{+\infty\}$. Since from now on, we will only talk about singleton-loop graphs, we will only write loops instead of singleton loops.

The basic idea of the algorithm is transform the CFG into an directed acyclic graph, for which we can easily compute the longest path. To this end, we need to identify the innermost loop, compute the longest path within this loop, replace it by an artificial node and adapt the edge weights (to account for the runtime within the loop). See Figure 3 and Figure 4 for an example.

$LPS(G, s) :=$

1. identify the loops $(L_j)_{j \in \{1,\ldots,l\}}$ of G by computing the strongly connected components.
2. for each $L_j = (V_{L_j}, E_{L_j})$:
   (a) modify L_j by replacing $\mathscr{E}(L)$ by two nodes $\mathscr{E}_{\mathrm{out}}$ and $\mathscr{E}_{\mathrm{in}}$ and by replacing all incoming edges $(v, \mathscr{E}(L))$ by edges $(v, \mathscr{E}_{\mathrm{in}})$ and all outgoing edges $(\mathscr{E}(L), v)$ by edges $(\mathscr{E}_{\mathrm{out}}, v)$
   (b) call $LPS(L_j, \mathscr{E}_{\mathrm{out}})$
   (c) now we know the $\mathrm{lps}(L_j, \mathscr{E}_{\mathrm{out}}, v)$ for all $v \in V_{L_j}$, and thus we set
   $$\begin{aligned}\mathrm{lps}(G, \mathscr{E}(L_j), v) :=\\ (\mathrm{b}(\mathscr{E}(L_j)) - 1) \cdot \mathrm{lps}(L_j, \mathscr{E}_{\mathrm{out}}, \mathscr{E}_{\mathrm{in}})\\ + \mathrm{lps}(L_j, \mathscr{E}_{\mathrm{out}}, v),\end{aligned}$$
   that is the longest path weight from $\mathscr{E}(L_j)$ to v is to the loop $\mathrm{b}(\mathscr{E}(L_j)) - 1$ times through L and then to the head for v.
   (d) replace L_j in G by a single node r_j and add an edge (r_j, x) for each $(p, x) \in \mathcal{T}(L_j)$ with appropriate weights, namely:
   $$w(r_j, x) := lps(G, \mathscr{E}(L_j), p) + w(p, x))$$
   Add an edge (v, r_j) for each $(v, \mathscr{E}(L_j)) \in E$ and set $w(v, r_j) := w(v, \mathscr{E}(L_j))$.
3. the altered graph is a DAG, thus we can easily determine the longest paths.
4. compute the longest path weights to nodes within the loops: Replace the nodes r_j again by the corresponding loops and set for each $L_j = (V_{L_j}, E_{V_j})$ and for all $v \in V_{L_j}$:
   $$\mathrm{lps}(G, s, v) := \overline{\mathrm{lps}}(G, s, \mathscr{E}(L_j)) + \mathrm{lps}(L_j, \mathscr{E}(L_j), v)$$

So far, we have not yet discussed, how $\overline{\mathrm{lps}}(G, s, \mathscr{E}(L_j))$ in step 4 is computed. Note that the entry node $\mathscr{E}(L_j)$ corresponds to a contraction node c in the condensed graph G'. When we compute the longest path weight from s to c, we set $\overline{\mathrm{lps}}(G', s, c) := \max_{v \in \mathrm{inc}(c)} \mathrm{lps}(G', s, v) + w(v, c)$. An example on how the algorithm works is given in Figure 3 and Figure 4.

3.3 Running Time - Numeric Bounds

Let us first analyze the algorithm's running time $T(|V|, |E|)$ for the case in which all loop-bounds are numeric values. In step 1), the strongly connected components of G are computed, which can be done in $O(|V| + |E|)$ time by depth-first search. Step 2a) can be computed in $O(\deg(\mathscr{E}(L_j)))$ time. In step 2b), the algorithm is called recursively which takes $T(|V_{L_j}| + 1, |E_{L_j}|)$ time. The weight updates in 2c) can be performed in $O(|V_{L_j}|)$ and the updates in 2d) in $O(|\mathcal{T}(L_j)|)$ time. It is folklore that the computation of longest path weights in a DAG, as done in step 3), takes no more than $O(|V| + |E|)$ time. Finally, step 4 can be done in $O(|V|)$. Hence, without the recursive calls, we have a linear running time $O(|V| + |E|)$.

Note that the recursion depth of our algorithm is bounded by $|V|$, as a node is split at most once. Furthermore the edge set of the sub-loops are disjoint. Although nodes are split, we can argue that the total number of nodes in a certain recursion depth is bounded

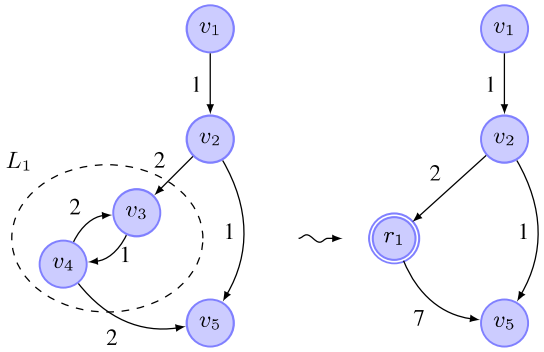

Figure 3. We assume the loop bound $\mathrm{b}(v_3)$ at entry node v_3 to be 3 and that v_1 is the source node s. The algorithm proceeds as follows: (a) the strongly connected component L_1 is identified (b) the longest path weight of 7 from v_3 to v_4 with respect to b is computed recursively (c) L_1 is replaced by r_1 (d) the longest path weights are computed in the resulting DAG.

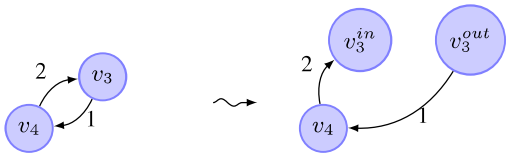

Figure 4. Entry node v_3 of the strongly connected component L_1 is split into v_3^{in} and v_3^{out}. The longest path weight from v_3^{out} and v_3^{in} is 3. With loop bound $\mathrm{b}(v_3) = 3$, this path is taken 2.

by $2|V|$ as follows: Let

$$V^{out} = \{v \in V \mid v \text{ has at least 1 outgoing edge}\}$$

and

$$V^{in} = \{v \in V \mid v \text{ has at least 1 incoming edge}\}$$

Then $\sum_{L \in \mathrm{sloops}(G)} |V_L^{out}| \leq |V^{out}|$ and $\sum_{L \in \mathrm{sloops}(G)} |V_L^{in}| \leq |V^{in}|$, where V_L is the set of nodes of L after splitting the entry node. Hence the running time is bounded by

$$O(|V|(|V^{out}| + |V^{in}| + |E|)) = O(|V||E|)$$

3.4 Running Time - Symbolic Bounds

In the presence of symbolic loop-bounds we have to change our algorithm slightly. Instead of a unique longest path we now have to consider for each target node a set of longest paths to that node (see Figure 5 for an example). When concatenating two paths we now have to concatenate all pairs of paths. Since the operations on the path weights include multiplications and additions, they can be represented as polynomials over the symbolic loop-bounds. Clearly, we aim at getting all possible path weights that are maximal for at least one choice for the symbolic loop-bound parameters. On the down side, testing whether a path weight is maximum for some choice (or instantiation) of the parameters seems to be non trivial. A compromise is to keep all paths with weights that are not dominated by another weight (i.e. all coefficients in the weight polynomial are at least as big as the coefficients in the other weight polynomial) to keep the solution set sparse in practice, which can be implemented very efficiently. Furthermore, as we will see later, for a some certain class of CFGs this step is necessary and sufficient to compute a minimal number of paths. For a problem instance $I = (G, s, t)$, consisting of a graph, a source node s and a destination node t, we denote by $\mathcal{D}(I)$ (or short $\mathcal{D}(s, t)$, if G can be deduced from the context) the set of longest path weights from s to t computed by our algorithm. The property of $\mathcal{D}(I)$, that its elements are pairwise non-dominating can be achieved by eliminating dominated elements after the execution of step 2c. We write $\mathrm{slbs}(I)$ for the number of symbolic loop-bounds of a problem instance $I = (G, s, t)$ and $\mathrm{lbs}(I) := \mathrm{lbs}(G) := |\mathrm{iloops}(G)|$ for the number of induced loops of G.

Theorem 1. *The algorithm's running time is polynomial in the input size and in the size of the output.*

Proof. First note that the running time only changes for the parts of the algorithm in which calculations on path weights are performed, namely the parts 2c), 2d) and 4). We will restrict this proof to the operations involved in step 2c), since the number of operations involved in 2c) is certainly not smaller than the ones in 2d) and 4).

Let us first count the number of operations on weight polynomials. Consider a longest path P from the source node s to the destination node t. Let $\mathrm{lps}(u, v)$ denote the longest path weights, computed by the algorithm for the longest paths from node u to node v, then for each loop $L = (V_L, E_L) \in \mathrm{loops}(G)$ that is traversed by P, we have $|\mathrm{lps}(\mathscr{E}(L), p_L)| \leq |\mathrm{lps}(s, p_L)|$ for $p_L \in \mathcal{P}(L)$ over which P leaves L again. Furthermore, for each $p_L \in portals(L)$ we have $O(|\mathrm{lps}(\mathscr{E}(L), \mathscr{E}(L))| \cdot |\mathrm{lps}(\mathscr{E}(L), p_L)|)$ operations, since the addition involves the addition of all pairs of weights in $\mathrm{lps}(\mathscr{E}(L), \mathscr{E}(L))$ and in $\mathrm{lps}(\mathscr{E}(L), p_L)$. Since L is strongly connected, $|\mathrm{lps}(\mathscr{E}(L), \mathscr{E}(L))| \leq |\mathrm{lps}(\mathscr{E}(L), p_L)|$. Thus the number of operations is bounded by $|\mathrm{lps}(\mathscr{E}(L), p_L)|^2 \leq |\mathrm{lps}(s, p_L)|^2$. Since each node in V_L can be a portal node of L, the total number of operations on polynomials occurring on the first recursion level is bounded by $\sum_{v \in V} |lps(s, v)|^2 \leq \left(\sum_{v \in V} |lps(s, v)|\right)^2$. But, since $\sum_{v \in V} |lps(s, v)|$ is just the number of path weights, reported by the algorithm, the number of operations on polynomials is polynomial in the number of reported path weights. Note that each weight has a unique representation and that all operations on the weight polynomials can be carried out in time polynomial in the size of these polynomials.

What is left to show is, that the weight polynomials computed for the nodes in the input graph have a size polynomially bounded by the size of the weight polynomials that are reported by our algorithm, that is the weight polynomials of the longest paths from the source node s to the sink node t. We will use a structural induction over the input graph G to prove so. If G contains no loops, the claim is true since G must be a DAG and therefore, all computed longest path weights are just constants. So, let us assume that G contains loops. By induction hypothesis, the claim holds now for each problem instance $(L, \mathscr{E}_{\mathrm{out}}, p)$ where L is a loop of G, where the entry node of L is split into the nodes $\mathscr{E}_{\mathrm{out}}$ and $\mathscr{E}_{\mathrm{in}}$ and where p is an arbitrary portal node of L. But then the claim is also true for $(L, \mathscr{E}(L), p)$ what can be seen as follows: Recall that a longest path weight from $\mathscr{E}(L)$ to p is given by the equation

$$\mathrm{lps}(G, \mathscr{E}(L), p) = (\mathrm{b}(\mathscr{E}(L)) - 1) \cdot \mathrm{lps}(L, \mathscr{E}_{\mathrm{out}}, \mathscr{E}_{\mathrm{in}}) + \mathrm{lps}(L, \mathscr{E}_{\mathrm{out}}, p)$$

for some path weights $\mathrm{lps}(L, \mathscr{E}_{\mathrm{out}}, \mathscr{E}_{\mathrm{in}})$ and $\mathrm{lps}(L, \mathscr{E}_{\mathrm{out}}, p)$. But then, $\mathrm{lps}(G, \mathscr{E}(L), p)$ is as least as large as the maximum of the sizes of $\mathrm{lps}(L, \mathscr{E}_{\mathrm{out}}, \mathscr{E}_{\mathrm{in}})$ and of $\mathrm{lps}(L, \mathscr{E}_{\mathrm{out}}, p)$ as each term in $\mathrm{lps}(L, \mathscr{E}_{\mathrm{out}}, \mathscr{E}_{\mathrm{in}})$ appears with a multiple of $\mathrm{b}(\mathscr{E}(L))$, $\mathrm{lps}(L, \mathscr{E}_{\mathrm{out}}, p)$ does not contain $\mathrm{b}(\mathscr{E}(L))$ and each term in $\mathrm{lps}(L, \mathscr{E}_{\mathrm{out}}, p)$ can eliminate only terms that are not multiplied with $\mathrm{b}(\mathscr{E}(L))$. The last thing we have to show now is, that the claim holds for (G', s, t), where again G' denotes the condensed graph. We compute the longest path weights in the directed acyclic graph G' by the recurrence $\mathrm{lps}(G, s, u) = \max_{v \in \mathrm{inc}(u)} (\mathrm{lps}(G, s, v) + w(v, u))$

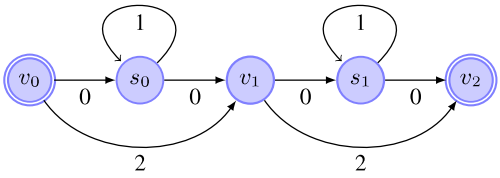

Figure 5. The different weights for the longest paths from v_0 to v_2 are 4, $2 + \mathrm{b}(s_0)$, $2 + \mathrm{b}(s_1)$ and $\mathrm{b}(s_0) + \mathrm{b}(s_1)$

starting with $u := t$. Consider now a weight polynomial $P = \mathrm{lps}(G, s, v) + w(v, u)$. Since we consider the condensed graph G', $w(v, u)$ is a polynomial containing only variables associated with the loop that in turn is associated with the node v (in the case that v is not a condensed node, $w(v, u)$ is just a constant). Thus, except for the constant terms, P contains at least as many terms as there are in $\mathrm{lps}(G, s, v)$ or in $w(v, u)$, which completes the proof. □

Theorem 1 states that the runtime of our parametric path analysis is polynomial in the input and output size. Hence, the number of reported paths influences the actual runtime. Unfortunately, there are examples such that the number of distinct formulas is exponential in the number of loop bounds (see [1] for further details). For each of these paths, there is an parameter instantiation such that this path leads to the worst-case execution time bound. Note that each approach to parametric path analysis must report this many path. In practice, however, both the number of parametric loop bound and the number of distinct reported path is quite limited as we will see in Section 5.

3.5 Correctness

Now, we will show that our algorithm indeed computes the weight of a longest valid path $\mathrm{lps}(G, s, t)$ from a source vertex s to a destination vertex t. In the following, when talking about paths we always mean valid paths. Again we will assume that G is a singleton-loop graph with weight function $w\colon E \mapsto \mathbb{N}$ and that we are given a loop-bound function $\mathrm{b}\colon \mathscr{E}(G) \to \mathbb{N} \cup \{+\infty\}$. We will show the claim by induction over the recursion-level of the algorithm. If we assume that G contains no loops, G must be a directed acyclic graph and thus, our algorithm is correct. So, now assume that G contains loops. The induction hypothesis tells us now that for all recursive calls of our algorithm, we obtain correct results. Let $p := s \rightsquigarrow t$ be a longest path in G. Let us assume w.l.o.g. that p shares at least one node with a sub-loop of G, i.e. for some $L := (V_L, E_L) \in \mathrm{sloops}(G)\colon \mathscr{E}(L) \in V_L$. Thus p can be written as $p = s \rightsquigarrow p' \rightsquigarrow t$ with $p' = (\mathscr{E}(L) =: v_0, v_1, \ldots, v_k)$ such that $v_i \in V_L$ and k is maximal. Since any sub-path of a longest path must be again a longest path between its starting- and end-node (with respect to the validity), we have that $w(p') = \mathrm{lps}(G, \mathscr{E}(L), v_k)$. Consider now the condensed graph $\overline{G}$ obtained by replacing loop L by a node r as described in the algorithm. Then the path $s \rightsquigarrow r \to v_k \rightsquigarrow t$ is valid and has weight $w(p)$. Therefore, $w(\mathrm{lps}(G, s, t)) \leq w(\mathrm{lps}(\overline{G}, s, t))$. On the other hand, $w(\mathrm{lps}(G, s, t))$ cannot be strictly less than $w(\mathrm{lps}(\overline{G}, s, t))$, because otherwise there would be a path in $\overline{G}$ with weight strictly greater than a longest path in G, which also would not traverse L, since the weights of these paths are unequal. But this would mean that there is also a path in G—just bypassing L—with the weight $w(\mathrm{lps}(\overline{G}, s, t))$, which leads to a contradiction.

What is left to show is that our algorithm computes correct values for $\mathrm{lps}(G, \mathscr{E}(L), v_k)$. Let $p = (\mathscr{E}(L) =: v_0, v_1, \ldots, v_k)$ with $v_i \in V_L$ be a longest path. We can assume that p contains exactly $\mathrm{b}(\mathscr{E}(L))$ (respectively $\mathrm{b}(\mathscr{E}(L)) + 1$ if $v_k = \mathscr{E}(L)$) times the node $\mathscr{E}(L)$, otherwise we could extend the path by the path $v_k \rightsquigarrow \mathscr{E}(L) \rightsquigarrow v_k$ without violating validity.

Now each sub-path p' of p with $p' = \mathscr{E}(L) \rightsquigarrow \mathscr{E}(L)$ must have the same weight, since otherwise, by replacing the lower weight sub-path by the corresponding higher weight sub-path, we could obtain a path with higher weights. Thus, we can assume that there exists a longest path $\mathscr{E}(L) \overset{p'}{\rightsquigarrow} \mathscr{E}(L) \overset{p'}{\rightsquigarrow} \cdots \overset{p'}{\rightsquigarrow} \mathscr{E}(L) \overset{p''}{\rightsquigarrow} v_k$ with weight $(\mathrm{b}(\mathscr{E}(L)) - 1) \cdot w(p') + w(p'')$. Since p' and p'' must be longest paths, we are left to show that our algorithm computes the weights $\overline{\mathrm{lps}}(G, \mathscr{E}(L), \mathscr{E}(L))$ and $\overline{\mathrm{lps}}(G, \mathscr{E}(L), v_k)$ correctly. But this follows directly by the way we alter the loop L, i.e. by splitting the entry node of L into the two nodes $\mathscr{E}_{\mathrm{out}}$ and $\mathscr{E}_{\mathrm{in}}$. Since L was a loop, every node in L is reachable from $\mathscr{E}_{\mathrm{out}}$. By induction hypothesis the algorithm now computes recursively the right values, where obviously $w(\mathrm{lps}(\overline{L}, \mathscr{E}_{\mathrm{out}}, \mathscr{E}_{\mathrm{in}})) = w(\overline{\mathrm{lps}}(G, e, e))$. Which finishes this proof.

3.6 Transformation of non Singleton-Graphs

This far, our new approach is applicable to singleton graphs only. We assume that each loop has a unique entry node. To analyze non-singleton graphs, we must obtain this property. To this end, we duplicate the loop-body and redirect loop entry edges, such that each loop body has a unique entry node; see Figure 6 for an example.

This transformation comes at the cost of an increased runtime: Although the complexity of the algorithm itself remains unchanged, the input-size increases. In how far this influences the actual performance of the algorithm strongly depends on the analyzed control-flow graph. We will discuss this point in the Section 5.

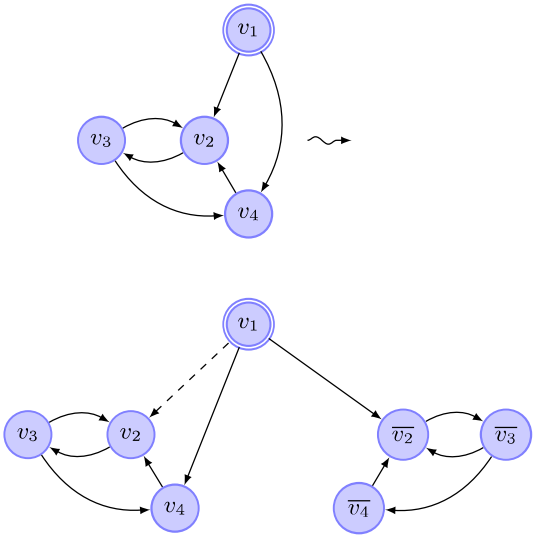

Figure 6. Transformation of non-singleton loops into singleton loops.

4. Handling Control Flow Constrains

An advantage of the implicit path enumeration technique is given by the possibility to define further constraints on the control flow. A typical example is the exclusion of infeasible path; different conditions of the program to be analyzed might depend on each other, such that certain control flow paths can be excluded. These cases can be coded using control flow constraints

One way to handle such constraints in our approach is to assign boolean literals to the edges of the control-flow graph that

indicate whether an edge is part of the graph or not. The goal is now to determine a longest path over all possible truth assignments of the boolean variables occurring in the deadpath constraints. A naive way of doing so, would be to solve the problem from scratch for each possible assignment and to compare in retrospect the obtained longest paths. The obvious drawback of this approach is that in any case an exponential number of problems have to solved. One possible improvement rests upon the observation that in different calls of our algorithm, the computation of longest subpaths are potentially repeated —independent of the given truth assignment. Thus, instead of computing the longest paths for each truth assignment, we just perform a single call in which we bookkeep for each computed subpath the literals that occurred on that path. At the point in our algorithm, when two paths are concatenated, we first check, whether the literals occurring in both paths are compatible, i.e. whether there can be a truth assignment that allows a path to consist of these two subpaths, which is clearly not the case if in the first subpath there is an edge with a variable that occurs in its negation in the second subpath. Once our algorithm has finished, we are left with a set of longest paths together with a set of boolean literals. In the final step, we examine these paths for all possible truth assignments. Since this set should be significantly smaller in practice than the union of all sets computed in the naive way, this should lead to significant speed up.

Note that this extension is not yet implemented but planned as future work.

5. Evaluation

In this section, we evaluate the improvement of our new approach in terms of runtime and precision. In the following, we refer to our new method as the *Silent (SIngle-Loop ENTry)* method.

The main motivation of this work is parametric path analysis. Prior parametric approaches suffer from poor runtime and precision and hence, strongly limit the applicability of the parametric timing analysis. Nevertheless, we also evaluate the performance in case of numeric path analysis to see how it compares to the state-of-the-art path analysis IPET (both, based on a commercial and a free ILP-Solver).

The Silent method exploits the often rather regular structure of software for safety-critical embedded systems. Hence, we first determine how many of our test-cases exhibit only singleton loops, i.e. fit our model. We then distinguish between the performance for singleton graphs and non-singleton graphs

Regarding the precision, the new method does not suffer from any inherent overapproximation and thus, computes exact results even in the parametric case. The loop-bound conversion from relative to absolute loop-bounds as in [2] is not needed. Regarding the precision, we compare Silent to implicit path enumeration using PIP [8].

5.1 Test Setting

All programs have been compiled via gcc to the ARM7[1] target architecture, one of the most eminent processors for embedded systems. The binary-files determine control-flow graphs, which are the input to both, Silent and IPET. Note that timing analysis has to resort to the binary level. Compiler transformation may influence the control flow extracted from high-level source files. All tests have been performed on an Intel Core2Duo, 2GHz, 2 GB Ram with Ubuntu 9.10 operating system. The execution times were measured using the unix-command *time* and thus, include in all cases time for the algorithm and all input/output operations. Since most execution times are small (time < 5 seconds), we repeated the tests 10 times and present the average.

5.2 Structure of the Benchmarks

The most eminent benchmark suite in the area of timing analysis for hard real-time systems is the Mälardalen WCET benchmark suite[2].

Analyzing executables instead of source-code files has several consequences. Calls to external library functions may be implicit in the source-code, but contribute to the complexity of the executable's control-flow graph. Hence, the analyzed control-flow graph and control-flow of the source-code may differ. Even if the source code contains only singleton loops, the corresponding executable may exhibit non-singleton loops. In addition, complex compiler optimizations—although not common for safety-critical systems—transform the control-flow graph.

An overview of the actual structure of the benchmarks is given in Table 1. Beneath the sizes of the source-code and of the executable of the benchmarks, it shows which benchmark fits into the singleton graph model and, if not, how often loops need to be duplicated. Note that this number does not refer to the number of non-singleton loops, but to the number of duplicated loops in the transformed control-flow graph. If a loop has n loop entries, it needs to be duplicated n-times. As Table 1 shows, only 8 of 33 test-cases

Name	Size (in Byte)		Singleton	# duplicated
	C-File	Exec	Graph	loops
adpcm	26582	156759	no	5
bs	4248	144447	yes	-
bs100	2779	144629	yes	-
cnt	2880	149801	yes	-
compress	13411	149804	no	65
cover	5026	148301	yes	-
crc	5168	145615	yes	-
duff	2374	144739	no	6
edn	10563	150682	yes	-
expint	4288	145867	yes	-
fac	426	144148	yes	-
fdct	8863	147128	yes	-
fft1	6244	153303	yes	-
fibcall	3499	144152	yes	-
fir	11965	151589	yes	-
insertsort	3892	144305	yes	-
jannecomplex	1564	144242	yes	-
jfdctint	16028	146858	yes	-
lcdnum	1678	144509	yes	-
lms	7720	157868	yes	-
ludpcm	5160	151848	yes	-
matmult	3737	145083	no	4
minver	5805	152845	yes	-
ndes	7345	148689	no	2
ns	10436	149567	no	7
nsichneu	118351	176240	yes	-
prime	904	144538	yes	-
qsort-exam	4535	146468	no	3
qurt	4898	151214	yes	-
recursion	620	144341	yes	-
select	4494	146283	yes	-
sqrt	3567	154282	yes	-
statemate	52618	162879	no	7

Table 1. Benchmarks; Sizes (of source code and of executables) and structure.

[1] `http://www.arm.com/products/CPUs/families/ARM7Family.html`

[2] `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

exhibit non-singleton loops, only in one case (*compress*) a higher number of loop-duplications is needed. Deeply nested loops, unstructured code segments as well as calls to external library functions are the reasons for non-singleton loops. The benchmark *compress* with 65 duplicated loops exhibits goto-statements and nested loops, *ns* and *statemate* contain nested loops.

Name	Size (in Byte)		Singleton	# duplicated
	C-File	Exec	Graph	loops
s-graph-1	208273	235222	yes	-
s-graph-2	468944	292305	yes	-
s-graph-3	702670	386961	yes	-
s-graph-4	936396	481609	yes	-
s-graph-5	670452	284593	yes	-
ns-graph-1	90274	215433	no	77
ns-graph-2	315562	247443	no	77
ns-graph-3	766144	426427	no	77
ns-graph-4	990502	520579	no	5
ns-graph-5	979908	518338	no	9
ns-graph-6	942084	502580	no	74

Table 2. Artificial Benchmarks; Sizes (of source code and of executables) and structure

Although Mälardalen Benchmark Suite is most common suite for timing analyses, the test-cases are rather small. In fact, all benchmarks are solved in less than one second by IPET and Silent. Unfortunately, other free benchmarks suites (Mibench, Papabench) for real-time systems contain academic examples of limited size, too.

Especially for the numeric performance evaluation, larger examples are needed. To this end, we created new benchmarks by combining and duplicating original test-cases from the benchmark suite. We also want to classify the new artificial benchmarks depending on whether or not they exhibit non-singleton loops. So, we only combined test-cases from the same group. Benchmarks *s-graph-x* are combinations of singleton graph examples, *ns-graph-x* of non-singleton graph examples. Table 2 presents the increased benchmarks.

Note that code duplication and combination of different files to increase the benchmarks does not necessarily increase the number of duplicated non-singleton loops in the graph. If, for instance, a library routine exhibits such a loop, code duplication just leads to more call sites, but not to more such loops. Although we applied this very simple approach to create larger test-cases, Table 3 later on shows that it suffices to lead to strong variation in the execution time.

5.3 Numeric Path Analysis

In the performance evaluation, we distinguish between singleton and non-singleton graphs.

Name	Runtime (s)		
	Silent	lp_solve	CPLEX
nsichneu	0.02	0.86	0.05
s-graph-1	0.03	3.46	0.08
s-graph-2	0.05	13.69	0.08
s-graph-3	0.08	30.85	0.11
s-graph-4	0.11	57.31	0.18
s-graph-5	0.11	108.8	0.13

Table 3. Performance evaluation, Numeric Analysis, Singleton Graphs

In case of singleton-loop graphs, Silent method strongly outperforms the prior path analysis technique using lp_solve and even competes with CPLEX. While lp_solve needs up to over 100 seconds for the larger benchmarks, Silent and CPLEX solve each test-case in about one second or less. The worst-case runtime of Silent is polynomial, runtime of CPLEX and lp_solve is exponential. CPLEX, however, implements a large variety of heuristics which enables a much better performance in practice.

Name	Runtime (s)		
	Silent	lp_solve	CPLEX
adpcm	0.04	0.07	0.02
compress	0.3	0.03	0.03
statemate	0.05	0.3	0.04
ns-graph-1	0.97	4.5	0.04
ns-graph-2	0.95	14.58	0.05
ns-graph-3	1.01	48.13	0.12
ns-graph-4	0.14	92.1	0.11
ns-graph-5	0.16	113.3	0.12
ns-graph-6	0.64	65.9	0.17

Table 4. Performance evaluation, Numeric Analysis, Non-Singleton Loop Graphs

In case of non-singleton loop graphs, loop duplication strongly degrades the performance of Silent. The more loops need to be duplicated, the worse the performance of the analysis becomes. Here, the analysis of *compress* with 65 loop duplications, for instance, takes much longer than the analysis of other benchmarks of comparable size. Although, CPLEX outperforms Silent, it can still compete with lp_solve.

The evaluation shows that the Silent method is an alternative to lp_solve, both for singleton graphs and non-singleton graphs. Especially for larger test-cases the difference is immense and Silent can save up to 99% runtime compared to the free ILP solver. Only for smaller test cases, the difference between both methods is less dramatic, but still very obvious. Silent can even compete with CPLEX for singleton graphs, which are, as Table 1 shows, more common. However, for non-singleton graphs, CPLEX is the best alternative.

5.4 Parametric Path Analysis

As discussed in Section 5, there are only a few other approaches to parametric timing analysis in general, and thus, also to parametric path analysis.

A direct comparison to these approaches is not possible. Our parametric timing analysis resorts to the binary level and thus, takes precise cache and pipeline effects into account. To the best of our knowledge, other parametric approaches [3–6, 15] analyze source-code only and assume a simplified hardware model. The difference in the hardware model leads to completely different precision and different input-files such that also the performance is incomparable.

In addition, only Bygde et al. [4] presented a complete, implemented toolchain and a sophisticated evaluation. But again, it is based on the source-code level, such that only an indirect comparison is possible.

In the following, we will thus compare Silent to the numeric analysis and to the results obtained by parametric implicit path enumeration technique via PIP (which has been formerly used).

5.4.1 Parametric Path Analysis – Performance

In the parametric case, precision and runtime are of interest. The Silent method can only be compared against PIP directly. PIP, however, exhibits an extremely poor performance. None of the 5 largest benchmarks from Mälardalen benchmark suite can be solved via PIP; out-of-memory errors are reported. These instances are still trivial for our new approach. Therefore, we compare the

runtime of the parametric path analysis to the numeric one. So, we can observe how the number of parameters and the number of reported formulas influence the runtime. Note that the number of parameters is usually quite limited in practice. Other approaches usually assume a limited number of parameters, too. Bygde et al. [4], for instance, considers cases with at most two parameters.

Name	Runtime # of parameters						
	0	1	2	3	4	6	8
s-graph-1	0.03	0.03	0.03	0.03	0.03	-	-
s-graph-2	0.05	0.05	0.05	0.05	0.05	0.05	0.06
s-graph-3	0.08	0.08	0.08	0.08	0.08	0.08	0.11
s-graph-4	0.11	0.11	0.11	0.11	0.12	0.12	0.12
s-graph-5	0.11	0.12	0.12	0.12	0.12	0.12	0.13
ns-graph-1	0.97	1	1.73	1.9	2.02	2.11	2.11
ns-graph-2	0.95	2.03	2.03	2.04	2.36	2.35	2.38
ns-graph-3	1.01	1.01	1.01	1.01	1.24	3.42	3.44
ns-graph-4	0.14	0.22	0.28	0.3	0.33	0.45	0.45
ns-graph-5	0.16	0.28	0.33	0.62	0.67	1.11	1.11
ns-graph-6	0.64	0.64	0.64	0.66	0.71	1.19	1.19

Table 5. Performance evaluation: Silent method, parametric test cases.

Table 5 shows that the parametric case exhibits an increased runtime compared to the numeric case. This increase ranges from a factor of $\approx$ 1 (singleton-graphs) to a factor of $\approx$ 5 (non-singleton graphs). Considering the small execution times of the Silent method, even such an increase is acceptable. Note that Bygde et al. also fail to compute parametric formulas in case of larger examples. Benchmark *nsichneu*, for instance, can not be solved using the minimum propagation analysis as presented in [4] due to the high complexity of their method.

Name	# of parametric formulas						
	1	2	3	4	6	8	
s-graph-1	2	2	2	4	-	-	
s-graph-2	2	4	8	16	16	16	
s-graph-3	2	4	8	16	16	32	
s-graph-4	2	4	8	16	32	64	
s-graph-5	2	4	8	16	32	64	
ns-graph-1	1	23	33	33	33	33	
ns-graph-2	13	13	13	17	17	17	
ns-graph-3	1	1	1	4	15	15	
ns-graph-4	12	21	21	21	25	25	
ns-graph-5	12	12	33	33	56	56	
ns-graph-6	1	1	1	2	8	8	

Table 6. Performance evaluation: Silent method, number of parametric formulas

The runtime in the parametric case is output-polynomial. Table 6 shows the number of parametric formulas reported by the Silent method. Although we can see that the output-size influences the performance, the number of duplicated loops still has a stronger impact in practice. The maximal number of reported formulas is 64 (*s-graph-4* and *s-graph-5*, 8 parameters, 0 duplicated loops)—runtime of the analysis in both cases, however is only about 10% of the runtime of *ns-graph-6*, with 8 parametric formulas, but 74 duplicated loops. Note that our algorithm does not perform any further simplifications on the reported formulas. Hence, some formulas may be dominated by others and can be neglected.

5.4.2 Parametric Path Analysis – Precision

The parametric bounds computed by Silent method are precise, i.e., instantiating the formula delivers the same results as the numeric analysis with annotated loop bounds. There is no overapproximation within Silent—in contrast to former methods.

Already rather simple control-flow structures cause the imprecision of the path analysis via PIP. Figure 7 shows two parallel loops. If we assume for instance that L1 has a parametric loop bound, the worst-case path as derived by PIP will always contain loop L1. Loop nesting leads in the same manner to an overapproximation.

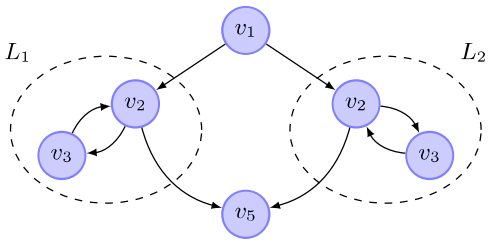

Figure 7. Parallel Loops

In the following, we present two simple benchmarks form Mälardalen benchmark suite, for which PIP was able to derive a parametric timing formula: *insertion-sort* and *matrix-multiplication*.

Insertion Sort

This benchmarks implements insertion sort. It contains one loop to initialize an array of size n and then sorts this array using a nested loop of depth 2.

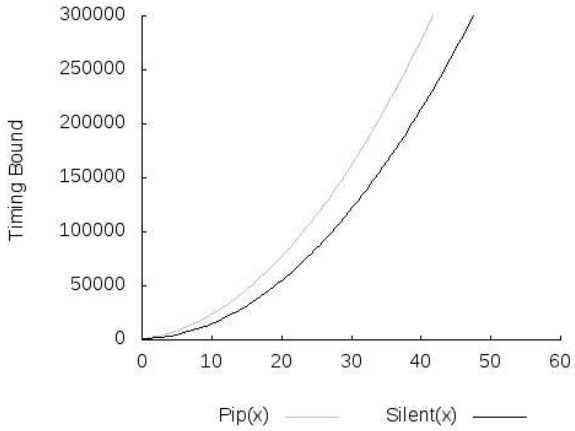

Figure 8. PIP vs. Silent—Insertionsort

$$Time_{PIP}(n) = 156n^2 + 674n + 1186$$
$$Time_{Silent}(n) = 131n^2 + 71n + 1185$$

Matrix Multiplication

This benchmark first initializes two $n \times n$ matrices and then multiplies them in $O(n^3)$ using nested parametric loops of depth 3.

$$Time_{PIP}(n) = \begin{cases} 386n^3 + 782n^2 \\ \quad +790n + 643 & \text{if } n > 1 \\ 2992 & \text{if } n \leq 1 \end{cases}$$

$$Time_{Silent}(n) = 111n^3 + 164n^2 + 845n + 793$$

In both benchmarks, the coefficients in $Time_{PIP}$ are higher than in $Time_{Silent}$, such that the overapproximation grows as the parameters increase. The overapproximation of the PIP method is caused

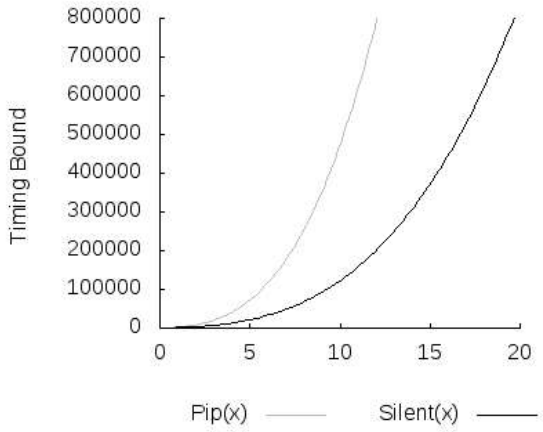

Figure 9. PIP vs. Silent—Matrix Multiplication

by loop nesting causes and the missing relation between execution counts of inner and outer loops. Note that since the parametric bounds computed by Silent method are precise, the shown difference in precision shows the imprecision of the path analysis via PIP.

6. Conclusions

We have developed a new path analysis able to derive numeric timing bounds as well as parametric timing formulas. Our new approach requires a special property of the programs to be analyzed: all loops in the control-flow graph have a unique entry node. We call such loops singleton loops. For all programs that contain only singleton loops, we can compute the worst-case path in polynomial time (in case of numeric loop bounds), resp., output-polynomial time (in case of parametric loop bounds) — in contrast to the exponential runtime of the former IPET approaches. Most test-cases from the Mälardalen Benchmark suite fit our model and even if non-singleton loops occur, we can still transform any control-flow graph such that the singleton loop property holds. As we have shown in the evaluation, such transformations may only increase the runtime slightly: Our new approach improves in practice upon prior parametric path analysis techniques in terms of precision and runtime. Also in the numeric case, we strongly outperform path analysis using implicit path-enumeration technique via lp_solve and can even compete with CPLEX. In the future, we plan to extend our method to parametric control flow constraints, to enable parametric deadpath-analysis as well as to allow arbitrary user annotations.

References

[1] E. Althaus, S. Altmeyer, and R. Naujoks. A new combinatorial approach to parametric path analysis. Reports of SFB/TR 14 AVACS 58, SFB/TR 14 AVACS, to appear.

[2] S. Altmeyer, C. Hümbert, B. Lisper, and R. Wilhelm. Parametric timing analyis for complex architectures. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08)*, pages 367–376. IEEE Computer Society, August 2008.

[3] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *25th IFAC Workshop on Real-Time Programming. Palma (Spain).*, May 2000.

[4] S. Bygde, A. Ermedahl, and B. Lisper. An efficient algorithm for parametric wcet calculation. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*, pages 13–21. IEEE Computer Society, August 2009.

[5] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Syst.*, 11(2):145–171, 1996.

[6] J. Coffman, C. A. Healy, F. Mueller, and D. B. Whalley. Generalizing parametric timing analysis. In *Proceedings of the 7th ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems (LCTES '07)*, pages 152–154, 2007.

[7] G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, NJ, 1963.

[8] P. Feautrier. The parametric integer programming's home `http:\www.piplib.org`.

[9] R. E. Gomory. An algorithm for integer solutions to linear programming. In R. L. Graves and P. Wolfe, editors, *Recent Advances in Mathematical Programming*, pages 269–302, New York, 1969. McGraw-Hill.

[10] R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, pages 26–30. IEEE Computer Society, 2004.

[11] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference (DAC '95)*, pages 456–461. ACM, 1995.

[12] B. Lisper. Fully automatic, parametric worst-case execution time analysis. In *Proceedings of the Third Internation Workshop on Worst-Case Execution Time Analysis (WCET 03)*, pages 77–80, July 2003.

[13] P. Puschner and C. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[14] H. Theiling. ILP-based Interprocedural Path Analysis. In *Proceedings of the Workshop on Embedded Software*, Grenoble, France, October 2002.

[15] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems (LCTES '01)*, pages 88–93. ACM Press, 2001.