Fully Automatic, Parametric Worst-Case Execution Time Analysis

Björn Lisper
Dept. of Computer Science and Engineering, Mälardalen University
P.O. Box 883, SE-721 23 Västerås, SWEDEN
bjorn.lisper@mdh.se

Abstract

Worst-Case Execution Time (WCET) analysis means to compute a safe upper bound to the execution time of a piece of code. *Parametric* WCET analysis yields symbolic upper bounds: expressions that may contain parameters. These parameters may represent, for instance, values of input parameters to the program, or maximal iteration counts for loops. We describe a technique for fully automatic parametric WCET analysis, which is based on known mathematical methods: an abstract interpretation to calculate parametric constraints on program flow, a symbolic method to count integer points in polyhedra, and a symbolic ILP technique to solve the subsequent IPET calculation of WCET bound. The technique is capable of handling unstructured code, and it can find upper bounds to loop iteration counts automatically.

1 Introduction

Real-time systems have timing requirements. These requirements imply upper limits on the execution times of codes in the systems: in order to guarantee that these requirements are met, it must be verified that the codes execute within their time limits. The purpose of *Worst-Case Execution Time* (WCET) analysis is to find upper bounds for the execution times of codes on given processors.

WCET analysis is usually divided into three parts: a fairly machine-independent *flow analysis* (or "high-level analysis") of the code, where information about the possible program flows is derived, a *low-level analysis* where the execution time for atomic parts of the code is decided from a performance model for the target architecture, and a final *calculation* where the information from these analyses is put together in order to derive the actual WCET bounds.

Three classes of calculation methods are mainly used. The *tree-based* approach is limited to well-structured codes, and assumes that the execution time bounds for programs can be directly derived from time bounds on their parts through simple rules [20]. Tree-based calculation methods are straightforward, but cannot utilize complex flow constraints well. *Path-based* techniques explicitly explore the execution paths of a program fragment [16, 24]. They can handle complex flow constraints somewhat better.

The *Implicit Path Enumeration Technique* (IPET), finally, models possible program flows for (possibly unstructured) control flow graphs with arithmetic constraints [19, 22]. Each entity i in the graph is given an execution time t_i by the low-level analysis. The WCET is estimated by $\max(\sum_i x_i t_i)$, where x_i is an execution count for entity i, subject to the constraints on program flow given by the flow analysis and the program flow graph. If these constraints are linear, then the optimization problem can be solved by integer linear programming (ILP) techniques. IPET is general and can handle complex flow constraints very well, but may be costly.

The basic IPET model assumes that the total execution time is independent of execution order. This holds only for simple CPU architectures. IPET has however been generalized to handle architectural features like pipelines [11] and caches [25]. For simplicity, we will only consider the basic IPET model here, but our method applies to the same problems as general IPET.

Parametric WCET analysis methods return a parameterized expression. These may be unknown parameters to the program, or represent unknown program flow information, like an unknown upper bound to a loop iteration count. In this paper we propose a systematic, general approach to parametric WCET analysis. Our proposed analysis can find parametric flow constraints, including loop bounds, automatically. The analysis works for code with arbitrary control flow graphs and can thus handle unstructured code. The parametric flow analysis is made by the *polyhedral abstract interpretation* by Halbwachs [10, 15], which represents sets of possible program states by polyhedra. Execution counts for statements can be restricted by the number of integer points in such polyhedra. Symbolic techniques for counting these points exist [6, 21]. A symbolic IPET calculation can then be performed, by *parametric integer programming* [12]. This is a symbolic ILP technique that finds parameterized optima.

The rest of this paper is organized as follows. The next section reviews related work. In Section 3 we give an introduction to abstract interpretation. Section 4 describes polyhedral abstract interpretation, and in Sections 5 and 6 we explain how it can be used to constrain the program flow. In Sections 7 and 8 we describe parametric integer programming and how to use it for a symbolic IPET calculation. Finally, in Section 9, we wrap up and give ideas for further research.

2 Related Work

Chapman [5] gives a method to compute regular expressions for paths yielding WCET formulae parameterized in unknown loop bounds. The method only works for reducible control flow graphs. Any non-structural program flow constraints (such as infeasible paths, constraints on loop bounds) must be entered by manual annotations.

A program representation for parametric WCET analysis has been suggested by Colin and Bernat [7]. This approach extends the classical program timing schema model [20]. It requires a well-structured high-level program, and constraints on program flow must be provided by hand. In [4] a similar, path-based approach is outlined. Vivancos et. al. [26] propose an iterative method for computing WCET for loops parameterized in the number of loop iterations, where a simple symbolic formula for the WCET of the loop is assumed and the loop timing behaviour is iterated until possible convergence. This method can take low-level features such as cache behaviour into account, but it is restricted to local analysis of loops. It cannot take global flow constraints into account. Parametric analysis of nested loops also seems to pose some problems.

Non-parametric, automated flow analyses have been considered. Healy et. al. [17] describe a method to bound the number of iterations in multiple-exit loops. The method demands a number of restrictions on loops in order to work. The method can also handle some cases of triangular-like nested loops, but similar restrictions apply.

Altenbernd [2] describes a symbolic execution method that can find infeasible paths automatically. The method puts restrictions on the program structure and requires that upper bounds for loop counts are given.

The Bound-T WCET tool [18] is sometimes capable of deriving bounds for loops automatically. Since this is a commercial tool details about the analysis are hard to get by, but we know that it uses Presburger Arithmetic. It works on machine code but requires a reducible loop structure.

Gustafsson [14] has developed a method that finds program flow information automatically by an interval-based abstract interpretation. The method is capable to find both upper

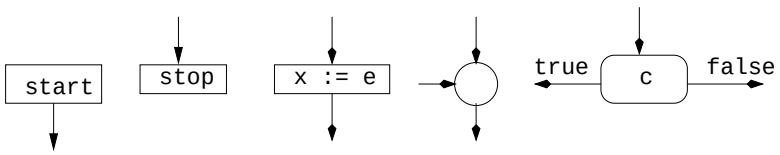

Figure 1: Flowchart nodes: start, stop, assignment, merge, and test.

bounds to loop counts and infeasible paths. It requires that unknown parameters are constrained to some fixed interval. The method works for unstructured code.

3 Abstract Interpretation

Abstract interpretation is a formal framework for program analysis. It guarantees *correctness*, in the sense that a predicted program property is surely true, and a generic solution method called *fixed-point iteration* will, under some conditions, yield the answer in finite time. The original framework by Cousot and Cousot [8] is defined for flowcharts (basically control flow graphs) describing imperative, possibly unstructured programs. A flowchart has a single *entry-* and *exit node*, respectively, *assignment nodes*, *test nodes*, and *merge nodes*, see Fig. 1. The framework can be extended to (possibly recursive) procedures and functions [9], but for simplicity we will stay with the flowcharts in this paper.

Each arc in the flowchart defines a program point. The state of the program, in a given program point, is an assignment of a value to each program variable. Many program properties can be expressed as properties of program states in certain program points. Abstract interpretation uses *abstract states*, which represent sets of program states. Each flowchart node is given an *abstract transition function*, which maps abstract states to abstract states.

The set of abstract states $\mathcal{S}$ must form a *complete lattice* $\langle \mathcal{S}, \sqcup, \sqcap, \sqsubseteq, \top, \bot \rangle$. $\sqsubseteq$ is an ordering relation corresponding to the subset relation on the corresponding sets of states. $\sqcup$, $\sqcap$ correspond to union and intersection, respectively, but may overapproximate the corresponding operation. Finally, $\bot$ is the *least element* (w.r.t. $\sqsubseteq$), representing $\emptyset$, and $\top$ is the *top element*, representing the universal set. If $S \sqsubseteq S'$, then S yields more precise information about the possible states than S'. The top element yields no information at all.

Program analysis by abstract interpretation assigns an abstract state S^p to each program point p, such that any reachable state there belongs to S^p. The abstract transition functions define a system of equations $\vec{S} = \vec{F}(\vec{S})$ for the abstract states. If certain axioms are met, then a solution can always be found by fixed-point iteration: $\vec{S}_i = \vec{F}(\vec{S}_{i-1})$. The iteration starts with the least possible assignment $\vec{S}_0 = \vec{\bot}$, assigning the bottom element to each program point, and will then produce the *least* assignment solving the equations. This gives the best possible precision in the program analysis.

For some abstract interpretations the fixed-point iteration will not always terminate. Termination can be guaranteed through a binary *widening operator* $\triangledown$ on abstract states, obeying some axioms [8]. For some program point p, the recursive equation $S^p_i = F^p(\vec{S}^{in}_{i-1})$ is changed to $S^p_i = S^p_{i-1} \triangledown F^p(\vec{S}^{in}_{i-1})$. If at least one equation along each cycle in the flowchart is changed in this way, then termination is ensured. The solution obtained when iterating the new equations will be correct, but it might not be the least one anymore. If the flowchart is reducible (contains well-structured loops only), then it suffices to insert widening after the merge point of each backedge.

4 Halbwachs' Polyhedral Abstract Interpretation

Halbwachs [10, 15] developed an abstract interpretation, for programs with numerical variables, where the abstract states represent polyhedra in a space whose coordinates correspond to the values of the different program variables. Polyhedra are equivalent to systems of linear inequalities: we will use them interchangeably below. Non-numerical variables can be dealt with by assuming that they are numerical but nothing is known about their values.

Thus, for a program with variables $x_1, \ldots, x_n$, an abstract state is a polyhedron P in an n-dimensional space or, equivalently, a system of linear inequalities $L\vec{x} \leq \vec{b}$ where $\vec{x} = (x_1, \ldots, x_n)$. ($\top$ corresponds to an empty system, without inequalities, and $\bot$ to a system without solutions.) The abstract operations are as follows:

- $\sqcap$ is intersection of polyhedra,
- $\sqcup$ is *convex hull*, that is: the smallest polyhedron enclosing the union of the two operands (see Fig. 2).
- $proj(P, x_i)$ is the least polyhedron enclosing P where all information about x_i is removed (see Fig. 2).
- The abstract transition function f for assignments $x_i := e$ has three cases:
  - e nonlinear: then $f(P) = proj(P, x_i)$,
  - $e = \vec{l} \cdot \vec{x} + b$, and $l_i \neq 0$: then $f(P) = P[x_i \leftarrow (x_i - \sum_{j \neq i} l_j x_j - b)/l_i]$, where $P[x_i \leftarrow t]$ is the polyhedron resulting from substituting t for x_i in the linear inequalities defining P. (Especially, $\top[x_i \leftarrow t] = \{x_i = t\}$ and $\bot[x_i \leftarrow t] = \bot$.)
  - $e = \vec{l} \cdot \vec{x} + b$, and $l_i = 0$: then $f(P) = proj(P, x_i) \wedge x_i = e$.
- For test nodes with condition c, there are transition functions f_{true} and f_{false} to the true/false-branch, respectively. We have the following cases:
  - c nonlinear: $f_{true}(P) = f_{false}(P) = P$,
  - $c \equiv \vec{l} \cdot \vec{x} = b$. If $P \subseteq (\vec{l} \cdot \vec{x} = b)$ then $f_{true}(P) = P$ and $f_{false}(P) = \bot$, otherwise $f_{true}(P) = P \sqcap (\vec{l} \cdot \vec{x} = b)$ and $f_{false}(P) = P$,
  - $c \equiv \vec{l} \cdot \vec{x} \leq b$: then $f_{true}(P) = P \sqcap (\vec{l} \cdot \vec{x} \leq b)$, and $f_{false}(P) = P \sqcap (\vec{l} \cdot \vec{x} \geq b + 1)$.
- merge node: then the transition function $f(P_1, P_2) = P_1 \sqcup P_2$.

See Fig. 2 for an explanation of some of the operations.

To ensure termination, Halbwachs defines a widening operator $\bigtriangledown$ such that $P_1 \bigtriangledown P_2$ essentially is the polyhedron obtained from P_1 by removing all constraints in P_1 violated by some point in P_2. Fig 2 illustrates this. For details, see [10, 15].

Linear inequalities can capture constraints that depend on several variables. Nested triangular loops can be handled accurately, as well as more irregular constraints given by linear conditionals within loops.

Let us consider a simple example: the flowchart in Fig. 3. It has nodes n_i, $i = 0, 2, 3, 4, 5, 8$, corresponding to basic blocks, and arcs (program points) numbered $0, \ldots, 10$. We assume each n_i has an execution time t_i. Each arc i has an execution count x_i, and the numbering is such that each basic block n_i has i as its preceding program point and thus the same execution count. We will use this simple program as a running example throughout the paper. We first analyze it in order to extract upper bounds for the execution counts for all statements. It suffices to analyze the program w.r.t. the possible values of i and n, since

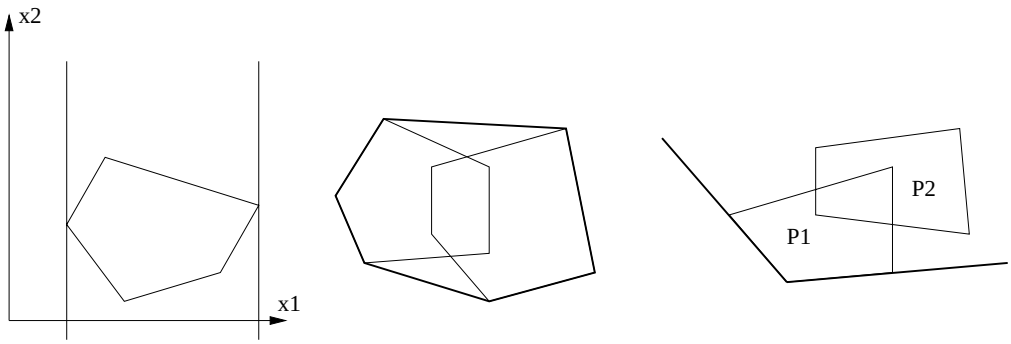

Figure 2: Some operations on polyhedra.

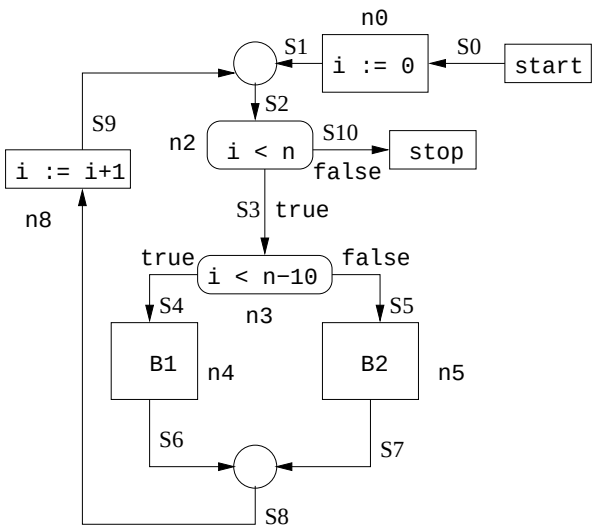

Figure 3: A simple flowchart program.

i	S_i^0	S_i^1	S_i^2	S_i^3	S_i^4	S_i^5	S_i^9	S_i^{10}
0	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
1	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
2	$\top$	$i = 0$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
3	$\top$	$i = 0$	$i = 0$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
4	$\top$	$i = 0$	$i = 0$	$i = 0$ $i \leq n-1$	$\bot$	$\bot$	$\bot$	$i = 0$ $i \geq n$
5	$\top$	$i = 0$	$i = 0$	$i = 0$ $i \leq n-1$	$i = 0$ $i \leq n-11$	$i = 0$ $n-10 \leq i \leq n-1$	$i = 1$ $i \leq n$	$i = 0$ $i \geq n$
6	$\top$	$i = 0$	$i \geq 0$	$i = 0$ $i \leq n-1$	$i = 0$ $i \leq n-11$	$i = 0$ $n-10 \leq i \leq n-1$	$i = 1$ $i \leq n$	$i = 0$ $i \geq n$
7	$\top$	$i = 0$	$i \geq 0$	$i \geq 0$ $i \leq n-1$	$i = 0$ $i \leq n-11$	$i = 0$ $n-10 \leq i \leq n-1$	$i = 1$ $i \leq n$	$i \geq 0$ $i \geq n$
8	$\top$	$i = 0$	$i \geq 0$	$i \geq 0$ $i \leq n-1$	$i \geq 0$ $i \leq n-11$	$i \geq 0$ $n-10 \leq i \leq n-1$	$i \geq 1$ $i \leq n$	$i \geq 0$ $i \geq n$
9	$\top$	$i = 0$	$i \geq 0$	$i \geq 0$ $i \leq n-1$	$i \geq 0$ $i \leq n-11$	$i \geq 0$ $n-10 \leq i \leq n-1$	$i \geq 1$ $i \leq n$	$i \geq 0$ $i \geq n$

Table 1: Fixed-point iteration for example flowchart.

they are the only variables affecting the program flow. We assume that B1 and B2 are basic blocks that update neither i nor n. There is one abstract state S^i for each program point i, however $S^6 = S^4$ and $S^7 = S^5$ since B1 and B2 touch neither i nor n. We use widening for S^2.

We assume no information about the values of i and n on entry: thus, $S^0 = \top$. The recursive equations then become:

$$\begin{array}{lcllcl} S_i^0 & = & \top & S_i^1 & = & \mathit{proj}\,(S_{i-1}^0, i) \wedge i = 0 \\ S_i^2 & = & S_{i-1}^2 \bigtriangledown (S_{i-1}^1 \sqcup S_{i-1}^9) & S_i^3 & = & S_{i-1}^2 \sqcap (i \leq n-1) \\ S_i^4 & = & S_{i-1}^3 \sqcap (i \leq n-10-1) & S_i^5 & = & S_{i-1}^3 \sqcap (i \geq n-10) \\ S_i^8 & = & S_{i-1}^4 \sqcup S_{i-1}^5 & S_i^9 & = & S_{i-1}^8[i \leftarrow i-1] \\ S_i^{10} & = & S_{i-1}^2 \sqcap (i \geq n) & & & \end{array}$$

Furthermore, it must hold that $S_i^8 = S_i^3$. The fixed-point iteration for the simplified system converges in nine iterations, see Table 1. Especially note the widening step $S_6^2 = S_5^2 \bigtriangledown (S_5^1 \sqcup S_5^9)$, which is crucial for convergence.

5 How to Derive Bounds for Execution Counts

How do we derive bounds on execution counts from abstract states? In a given program point i, S^i surely contains all the possible program states in that point. In all program points in a terminating, deterministic program, a new state must be reached every time the program point is encountered. Thus, for terminating programs, the number of elements in S^i bounds the execution count for i. The flowchart nodes with nonzero execution times are test and assignment nodes: these have a single in-arc and will therefore have the same execution count. It follows that for a terminating program, the abstract interpretation can derive bounds on the execution time for each node in the flowchart that corresponds to actual code.

However, in general not all variables affect the number of times a program point will be revisited. Thus, we really want to count only the number of different value combinations for the variables that may affect the control, that is: they appear in test nodes in the program fragment under analysis, or can affect the values of variables appearing there. This means to consider only polyhedra in a subspace spanned by such variables. Identifying

which variables might affect control can be done by *program slicing* [27] with respect to the conditions in the program. Program slicing removes all statements whose execution can surely not affect some given part of a program. The remaining program can then be analyzed. In our example, program slicing with respect to the conditions will remove the basic blocks B1 and B2.

An important kind of program fragment is loops. We identify all variables that appear in the test nodes in a loop. If the loop has single entry/single exit subgraphs, then the outcome of tests within the subgraph will not affect the program flow outside it. Thus, we don't need to consider the variables in the test nodes of such subgraphs when bounding the execution count for the outer loop.

Furthermore, some variables affecting the control should be regarded as parameters: namely, those that do not change while executing the program fragment under analysis. The number of points in a polyhedron limiting the execution count must be expressed as a formula that is parametric in these variables. Standard compiler techniques [1] can be used to find loop-invariant variables.

Finally, we must count the numbers of points in parametric polyhedra. Two techniques are known: through successive projection using known formulae for sums of powers of integers [21], and using *Ehrhart Polynomials* [6].

In the example in Fig. 3, the conditions depend on i and n only. n is never updated, and is thus considered a parameter. For each node, the number of elements in the abstract state on the preceding edge provides an upper bound on the execution count. The execution count for n_0 is trivially one. By the method in [21], we obtain:

$$\begin{aligned}
\left|S^2\right| &= \infty \\
\left|S^3\right| &= \left|S^8\right| = \mathit{if}\ \ n > 0\ \ \mathit{then}\ \ n\ \ \mathit{else}\ \ 0 \\
\left|S^4\right| &= \mathit{if}\ \ n \geq 11\ \ \mathit{then}\ \ n - 11\ \ \mathit{else}\ \ 0 \\
\left|S^5\right| &= \mathit{if}\ \ n \geq 1\ \ \mathit{then}\ \ (\mathit{if}\ \ n \geq 10\ \ \mathit{then}\ \ 10\ \ \mathit{else}\ \ n)\ \ \mathit{else}\ \ 0
\end{aligned}$$

This yields bounds $x_i \leq \left|S^i\right|$, where x_i is execution count for basic block n_i.

S^2 is overapproximated due to the widening. Therefore, there is no upper bound for x_2. This may seem awkward. But the structural flow constraints of the flowchart will ensure finiteness of x_2, see Section 7.

6 Constraints on Paths

Constraints on paths can be used to increase the precision of the calculated WCET bound. Given a path P between two program points p and q, we can calculate an abstract state S^q for q, approximating the set of states reachable along P from the states given by S^p in p, in the following way:

- form a flowchart F from the subgraph given by P, with an entry node immediately preceding p and an exit node immediately following q;
- assign the abstract state $\bot$ to all unconnected in-edges to merge nodes in F;
- select S^p as the abstract state for p given by the analysis of the whole program, and assign it to p in F (abstract starting state);
- solve the equations over F.

The number of elements in S^q bounds the number of times the path P can be taken, given that we start from some state in S^p. This information can be used directly in IPET calculations with pipeline timing effects [11]. The sum of the execution counts of two basic blocks in a loop can also be bound (infeasible path analysis). If p and q are in a loop with upper bound N for the iteration count, then the flow constraint $x_p + x_q \leq N + |S^q|$ is valid.

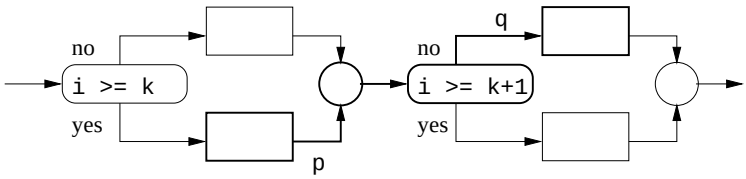

Figure 4: Loop body with partly infeasible paths.

The abstract transition functions for the original flowchart can be reused for F. If F is cycle-free, then S^q can be quickly computed from S^p without fixed-point iteration. A typical situation is when analyzing paths in an innermost, time-critical loop body.

In Fig. 4, an example of a loop body is given, with loop index i and parameters k, n. Say we want to compute a bound on $x^p + x^q$ given the loop index constraint $1 \leq i \leq n$. We then have $S^p = 1 \leq i \leq n \wedge k \leq i$. From this we can calculate

$$\begin{array}{rcl} S^q & = & (S^p \sqcup \bot) \sqcap i \leq k \\ & = & ((1 \leq i \leq n \wedge k \leq i) \vee \mathit{false}) \wedge i \leq k \\ & = & 1 \leq i \leq n \wedge i = k \end{array}$$

This provides the information that we can execute the path from p to q, for the given loop index range, only when $i = k$. We obtain the constraint $x^p + x^q \leq |1 \leq i \leq n| + |1 \leq i \leq n \wedge i = k|$. After counting integer points and simplifying, this yields

$$x^p + x^q \leq \mathit{if}\ \ n \geq 1\ \ \mathit{then}\ \ (\mathit{if}\ \ 1 \leq k \leq n\ \ \mathit{then}\ \ n+1\ \ \mathit{else}\ \ n)\ \ \mathit{else}\ \ 0.$$

7 Parametric Integer Programming

Parametric Integer Programming [12] solves the following symbolic ILP problem: find the optimum of an objective function over the set

$$\{\, \vec{x} \mid \vec{x} \geq \vec{0}, A\vec{x} + B\vec{z} + \vec{c} \geq \vec{0}, \vec{x} \text{ integral} \,\}.$$

Here, $\vec{z}$ is a vector of parameters, and the set is a function of this vector. The solution is a nested conditional expression, where the conditions are systems of linear inequalities in the parameter vector $\vec{z}$.

Contrary to ordinary ILP, the algorithm finds the *lexicographic* minimum [12], that is: the least vector in N^n w.r.t. the lexicographic order $\prec_n$ defined by:

$$(x_1, \ldots, x_n) \prec_n (x'_1, \ldots, x'_n) \iff \left\{ \begin{array}{l} x_1 < x'_1, \text{ or, for } n > 1, \\ x_1 = x'_1 \wedge (x_2, \ldots, x_n) \prec_{n-1} (x'_2, \ldots, x'_n) \end{array} \right.$$

However, it is easy to transform the problem of maximizing a linear objective function into an instance of this problem [13].

The algorithm is a symbolic extension of the dual simplex algorithm for linear programming augmented with Gomory Cuts [23] to ensure integral solutions. It will systematically split the parameter space and solve more constrained subproblems until unique solutions can be found with the given constraints on the parameters.

How can the algorithm be used to perform symbolic IPET calculations? The inequalities derived from the abstract interpretation contain formulae for the number of integer points in polyhedra. In general, these are nonlinear expressions in the parameters of the program. However, for each program point i, a new symbolic parameter s_i can be introduced for the expression for $\left|S^i\right|$, yielding linear constraints in lieu of the original, nonlinear constraint. In the final result, the formula for $\left|S^i\right|$ can be substituted back for s_i. A final simplification gives the symbolic WCET bound expressed in the input parameters of the

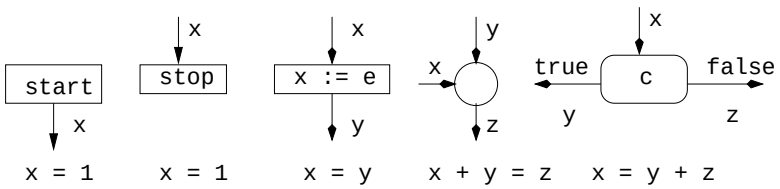

Figure 5: Structural flow constraints.

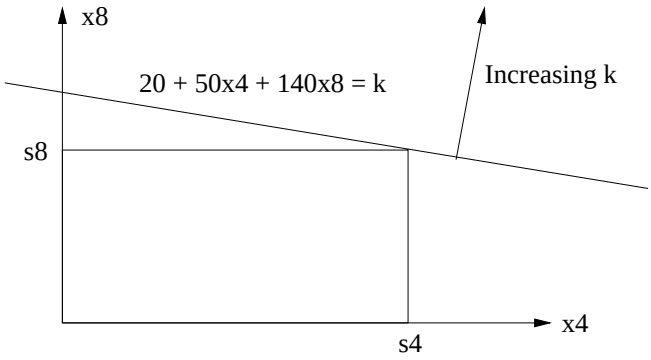

Figure 6: Maximizing the objective function in the IPET example.

program. An alternative is to derive a WCET bound expressed directly in symbolic execution count bounds. This can provide information about, for instance, the WCET sensitivity to certain loop counts.

This could potentially yield one symbolic parameter for each node in the flowchart, Many parameters give a risk of large resulting formulae and long execution time of the algorithm. Fortunately, the number of parameters can be reduced by considering the structural flow constraints of the flowchart. Each flowchart node generates an equality as shown in Fig. 5: these can be used to eliminate variables in the constraint system, and thus to reduce the number of needed symbolic parameters as well.

Let us now perform a parametric IPET calculation for the example in Fig. 3. The WCET estimate is $\max(\sum_{i=0,2,3,4,5,8} x_i t_i)$. The execution count bounds derived from the abstract interpretation yield constraints $x_i \leq s_i, \quad i = 1, \ldots, 10$, where s_i is a symbolic parameter for $\left|S^i\right|$. We also have non-negativity constraints $x_i \geq 0, \quad i = 1, \ldots, 10$.

Finally, the structural flow constraints can be used to reduce the number of variables down to two. Selecting x_4 and x_8 as basis yields the reduced problem $\max(t_0 + t_2 + x_4(t_4 - t_5) + x_8(t_2 + t_3 + t_5 + t_8))$ under the constraints $0 \leq x_4 \leq s_4$, $0 \leq x_8 \leq s_8$. Let us assume computation times $t_0 = 10$, $t_2 = 10$, $t_3 = 20$, $t_4 = 150$, $t_5 = 100$, and $t_8 = 10$. We then obtain the WCET estimate $50s_4 + 140s_8 + 20$ for $x_4 = s_4$, $x_8 = s_8$. See Fig. 6. With s_4, s_8 as the functions of n given by the polyhedral abstract interpretation we obtain (after simplification):

$$\begin{array}{rl} n \geq 11: & 190n - 530 \\ 0 < n \leq 10: & 140n + 20 \\ \text{otherwise}: & 20 \end{array}$$

8 An Observation about Calculation Methods

In the linear constraints considered in this paper (structural flow constraints, constraints on maximal execution count, path constraints), the unknown execution counts and parameters

all appear with coefficient 1, 0, or -1. The system matrix for the linear inequalities is then *unimodular*, which implies that all the corners in the corresponding polyhedron have integral coordinates [23]. It then suffices to use an ordinary linear programming algorithm to solve the ILP problem. The Parametric Integer Programming algorithm benefits directly from this, since it uses the dual simplex method as the first step. This observation also applies to conventional, non-parametric IPET.

IPET problems considered in the literature [19, 22] often have linear constraints with other coefficients. These typically arise from loops, encoding statements like "the loop body always executes at most c times the execution count of the basic block preceding the loop". This is a way to express an upper loop count bound c relative the environment of the loop. Our method calculates absolute upper bounds for each execution count, also inside nested loops, through the polyhedral abstract interpretation. This avoids the need for this kind of relative loop constraint. Constraints relative to, say, counters of enclosing loops are instead encoded in the polyhedra. They do not affect the calculation phase since the integer point counts of the polyhedra are replaced by symbolic parameters in the IPET calculation.

9 Conclusions and Further Research

We have demonstrated a fully automatic method for parametric WCET analysis that goes all the way from flow analysis to final WCET calculation. The method utilizes known methods, developed in other contexts, for parametric calculations. The resulting parametric WCET analysis is potentially very powerful. The method can automatically derive and solve both parametric constraints expressing absolute upper bounds on execution counts, and parametric path constraints. As far as we know, no other published method can accomplish this. A simple example demonstrates how it works.

The flow analysis and the calculation method are replaceable. Halbwachs' polyhedral analysis is capable of deriving parametric flow constraints, but other analysis methods are certainly possible, providing different tradeoffs between precision and computation time. A parameterized interval-based analysis is perfectly viable: it would give less precision at a lower cost. Conversely, one could design a flow analysis based on general Presburger Arithmetic: this would sometimes provide better precision but at a potentially very high cost. The parametric IPET calculation can handle any linear parametric constraints, regardless of the source.

The parametric IPET calculation method we propose generalizes conventional IPET calculation, and it interfaces to low-level analyses in the same way. The method is easily adaptable to other CFG variants than the flowcharts here. It needs static upper bounds to execution times for basic blocks. If these bounds are too pessimistic (due to architectural features), then the usual methods can be used, such as adding corrections for pipeline overlap [11] or virtually unfolding the first loop iteration in the CFG to obtain a more precise account for cache effects [25].

Future work involves a full implementation in order to evaluate the method with respect to accuracy and practical time complexity. We are currently implementing a tool for WCET analysis of full C [3]. The first version of this tool will use Gustafssons interval-based abstract interpretation [14] for flow analysis, and a conventional IPET calculation: to implement the parametric analysis in this context is an interesting future possibility.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] P. Altenbernd. *Timing Analysis, Scheduling, and Allocation of Periodic Hard Real-Time Tasks*. PhD thesis, Department of Mathematics and Computer Science, University of Paderborn, Germany, October 1996.

[3] N. Bermudo, J. Gustafsson, B. Lisper, C. Sandberg, and L. Sjöberg. A prototype tool for flow analysis of C programs. In G. Bernat, editor, *Proc. WCET 2002 Workshop*, Vienna, June 2002.

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

[5] R. Chapman. Worst-case timing analysis via finding longest paths in SPARK Ada basic-path graphs. Technical Report YCS246, The British Aerospace Dependable Computing System Centre, Dept. of Computer Science, Univ. York, Oct. 1994.

[6] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proc. International Conference on Supercomputing*, pages 278–285, Philadelphia, PA, 1996. ACM.

[7] A. Colin and G. Bernat. Scope-tree: a program representation for symbolic worst-case execution time analysis. In *Proc. 14th Euromicro Conference on Real-Time Systems*, Vienna, June 2002.

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

[9] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. J. Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 237–277. North-Holland, 1978.

[10] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.

[11] J. Engblom and A. Ermedahl. Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proc. 6^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, Dec. 1999.

[12] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Sept. 1988.

[13] P. Feautrier and N. Tawbi. Résolution de systèmes d'inequations linéaires; mode d'emploi du logiciel PIP. Internal report, Laboratoire MASI, Université P. et M. Curie, Paris, Dec. 1989.

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

[15] N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. Thèse de 3eme cycle, Univ. de Grenoble, Mar. 1979.

[16] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), Jan. 1999.

[17] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proc. IEEE Real-Time Applications Symposium (RTAS'98)*, June 1998.

[18] N. Holsti, T. Långbacka, and S. Saarinen. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, Sept. 2000.

[19] W. Li. Compiler cache optimizations for banded matrix problems. *Proc. 1995 Int. Conf. on Supercomputing*, July 1995.

[20] C. Park and A. Shaw. Experiments with a program timing tool based on a source-level timing schema. *IEEE Computer*, 24(5):48–57, 1991.

[21] W. Pugh. Counting solutions to Presburger Formulas: How and why. In *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 121–134, Orlando, FL, June 1994. ACM.

[22] P. Puschner and A. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, Jul. 1997.

[23] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

[24] F. Stappert, A. Ermedahl, and J. Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proc. 4^{th} International Workshop on Compiler and Architecture Support for Embedded Systems (CASES 2001)*. ACM Press, Nov. 2001.

[25] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *International Journal of Time-Critical Computing Systems*, 18:157–179, 2000.

[26] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric Timing Analysis. In J. Fenwick and C. Norris, editors, *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'2001)*, pages 88–93, Snowbird, Utah, June 2001.

[27] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.