An Efficient Algorithm for Parametric WCET Calculation

Stefan Bygde, Andreas Ermedahl and Björn Lisper
School of Innovation, Design and Technology
Mälardalen University, Sweden
Email: {stefan.bygde,andreas.ermedahl,bjorn.lisper}@mdh.se

***Abstract*—Static WCET analysis is a process dedicated to derive a safe upper bound of the worst-case execution time of a program. In many real-time systems, however, a constant global WCET estimate is not always so useful since a program may behave very differently depending on its configuration or mode. A *parametric* WCET analysis derives the upper bound as formula rather than a constant. This paper presents a new efficient algorithm that can obtain a safe parametric estimate of the WCET of a program. This algorithm is evaluated on a large set of benchmarks and compared to a previous approach to parametric WCET calculation. The evaluation shows that the new algorithm, to the cost of some imprecision, scales much better and can handle more realistic programs than the previous approach.**

I. Introduction

The worst-case execution time of a program is important to know in many real-time and embedded systems applications. With static analysis such a bound is found by analysing the source or object code and a model of the hardware. Most static WCET analyses derive a constant time estimation of the WCET of a program. This global WCET estimation correspond to the worst possible configuration and input to a program. A global WCET estimation is in many cases not so useful since timing often depends on modes or configurations of the analysed software [1], [2].

Parametric WCET analysis derives a WCET estimation as a formula rather than a constant. Such a formula contains more information about the WCET as it can be instantiated with known values to obtain concrete WCET bounds. Parametric WCET analysis is naturally more complex than classical WCET and is best suited to analyse small programs or functions which has a timing behaviour which is parametric in some variables. Interesting applications include real-time systems with disable interrupt sections, which are sections of code which are not allowed be interrupted. A case study [3] showed that these regions are typically small and particularly interesting to get a WCET estimation of. This study also points out the problem of having a constant, global WCET estimation. Component based software development [4], [5] is another interesting application where re-usable components designed to interact with each other in different contexts can be analysed in isolation. Since components are designed to function in different contexts, a reusable WCET estimation is desired. Component models designed for embedded systems (such as saveCCM [6] or Rubus [7]) typically uses quite small components which makes parametric WCET analysis interesting.

This work is supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS

This paper investigates the method proposed in [8] for obtaining WCET estimations which are parametric in some of a program's variables. The method is general and works for arbitrary program flows, i.e. it is applicable on unstructured code. The method is based on general program analysis techniques which easily can be adjusted for a precision/complexity trade-off.

In this paper, we present a new efficient algorithm for performing parametric calculation, which is an important part of the method in [8]. This method is much more efficient and scales better than the previously suggested method and it can analyse larger and more complex programs, this is to the cost of some precision loss.

The paper is organised as follows. A brief introduction to static WCET analysis with terminology is given in Section II. Section III gives an overview of the method presented in [8]. In Section IV we present the new parametric calculation. Section V evaluates the method by comparing it to the previously suggested one. Section VI presents related work in the field of parametric WCET analysis. Finally, Section VII concludes the paper and plans for future work are presented in Section VIII.

II. Static WCET Analysis

This section describes common concepts and terms used in static WCET analysis. An overview of the topic can be found in [9]. Static WCET analysis derives safe upper bounds of the worst-case execution time of a program. A *safe* bound means that it is guaranteed to be more than or equal to the real worst case. Typically WCET analysis is divided into three parts: flow analysis, low-level analysis and calculation. Flow analysis analyses the source or object code of a program to find constraints on the program flow. Examples of program flow constraints are the constraints coming from the structure of a program. Flow constraints can also be bounds on loop iterations, or infeasible paths. An infeasible path is a program path which due to semantic constraints will never be taken in practise. Low-level analysis uses a mathematical model of the hardware to estimate the worst-case timing for the basic blocks (or other atomic unit) of a program. A low-level analysis should include analysis complex hardware features such as

caches and pipelines (if present on the target platform), in order to be accurate. Finally, the calculation phase combines the results from flow and low-level analysis to calculate a concrete upper bound for the WCET. A common technique for WCET calculation is the Implicit Path Enumeration Technique (IPET) [10]. IPET estimates the WCET of a program P by maximising

$$\mathrm{WCET}_P = \max \sum_q c_q x_q$$

subject to some linear constraints, where c_q is the an upper bound of the worst case time of the block preceding program point q. These bounds are usually obtained by a low-level analysis. The factor x_q is the execution count of program point q, meaning the maximum number of times that it can be visited. The x_q variables are constrained by information obtained from flow analysis and from the structure of the program's Control Flow Graph (CFG). Maximisation is then obtained by integer linear programming.

III. Our Approach to Parametric WCET Analysis

This section gives a brief overview of the method introduced in [8], the work flow of the method is depicted in Figure 1. The method relies on a few well-known program analysis techniques which produces intermediate results which later contribute to a parametric WCET bound. We will briefly describe each box in Figure 1 and its role in the method.

1) Structural Analysis: Structural analysis builds the CFG and derives corresponding structural flow constraints of the program. Note that in this paper, the nodes of the CFGs correspond to statements rather than basic blocks.

2) Low-Level Analysis: As explained in Section II, the low-level analysis derives timing bounds for all atomic parts of the program.

3) Parametric Calculation: Parametric calculation is a parametric version of the calculation phase described in Section II. The idea of parametric calculation is to have a symbolic upper bound on the execution count for each program point. Parametric calculation then computes a WCET estimation expressed in terms of these symbolic bounds. The structural constraints and low-level analysis results are all that is needed to perform a parametric calculation. However, it is fully possible to add other constraints, such as loop bounds or infeasible paths, derived from flow analyses as well. In [8], the suggested approach to parametric calculation was Parametric Integer Programming (PIP) introduced by Feautrier in [11]. In this paper we will introduce a more efficient technique to perform parametric calculation, as described in Section IV.

4) Abstract Interpretation: The aim of the leftmost branch of Figure 1 is to provide more information about the symbolic execution bounds. Abstract Interpretation [12] is a common technique for abstracting program semantics. It can be used to derive invariants for the states at each program point. An example of such an invariant would be "variable x will always be between 3 and 5 at program point q". The nature of the invariants are chosen by a so-called *abstract domain*. A range of abstract domains have been suggested [13], [12], [14], [15], and they offer different trade-offs between precision and complexity. An interesting class of abstract domains are the *relational domains*, which preserve relations between program states and can express things as "variable x will always be less than $2y$ at program point q". The suggested abstract domain to use for parametric interpretation in [8] is the polyhedral domain [14], although other domains would also work.

5) Symbolic Counting: In a deterministic and terminating program each semantic state can be reached exactly once; given a state, the next one is uniquely given by the determinism, and the number of states has to be finite in a terminating program. By calculating invariants for each program point, for instance by abstract interpretation, we can often find an upper bound of the execution count of that program point. As an example, consider a program with two variables i and n. Assume that n is constant during the program, and that the invariant $0 \leq i \leq n$ holds in program point q (this information can be provided by abstract interpretation with a relational domain). Since n is constant, and there are no other variables, the determinism enforces that q can be visited at most $n+1$ times if n is non-negative. Thus, the bound on the execution count of q is parametric in the variable n.

6) Substitution: When we by abstract interpretation and symbolic counting have derived upper bounds on the program points expressed in terms of program variables, we can substitute these bounds for the symbolic ones in the parametric formula obtained by parametric calculation. This results in a formula parametric in program variables.

7) Simplification: Finally, after substitution, there might be a lot of redundant information in the final formula. Removing redundant information will result in a cleaner and more concise formula.

A. Example

Figure 2 shows an example program L. We assume, for simplicity, that each program point has a worst-case timing of 10 clock cycles (in practise, these timing should be provided by low-level analysis). The objective function for parametric calculation will therefore be

$$\sum_{q\ \in \mathcal{Q}_L} 10 x_q = 10 \sum_{q \in \mathcal{Q}_L} x_q$$

subject to the structural constraints and under the assumption that each execution count x_q has a symbolic upper bound p_q. Parametric IPET calculation (using PIP) will yield the following formula:

$$\begin{cases} 30p_k + 40 & \text{if } p_j, p_3 \geq 1 \wedge p_k \leq p_2 - 1 \\ 30p_2 + 10 & \text{if } p_j, p_3, p_2 \geq 1 \wedge p_k > p_2 - 1 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $p_j = \min(p_0, p_1)$ and $p_k = \min(p_4, p_5)$. This correspond to the parametric execution count formula in Figure 1.

Now we perform the abstract interpretation. Using the polyhedral domain [14], we obtain the following invariants, when applied on the example program L:

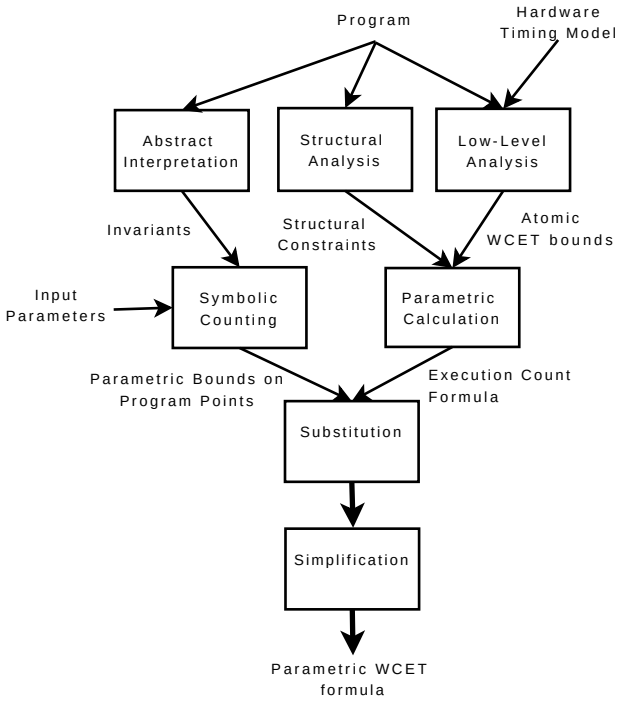


Fig. 1. Work flow

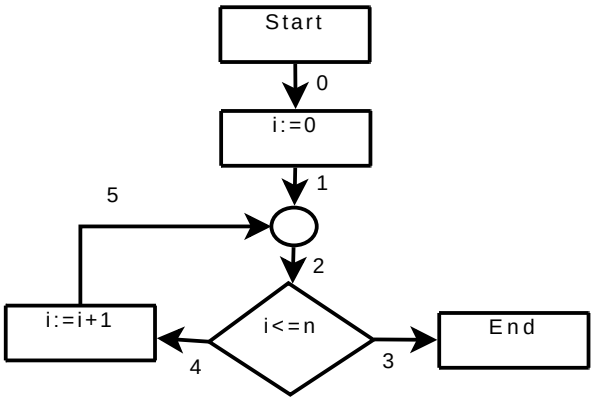


Fig. 2. Example program L with labelled program points

$$\begin{array}{ll} a_0 = \top & a_3 = \{i \geq 0, i \geq n+1\} \\ a_1 = \{i = 0\} & a_4 = \{0 \leq i \leq n\} \\ a_2 = \{i \geq 0\} & a_5 = \{1 \leq i \leq n+1\} \end{array}$$

where $\top$ means "no information". Next step (symbolic counting) is to compute execution bounds for the individual program points using this information. These bounds will be parametric in some input variables of the program. Suitable input-parameters are variables which are not changed during execution of the program and which affects program flow. In this case the variable n is a suitable choice.

In [16], Pugh suggests a method for symbolically computing the number of solutions to a Presburger formula. Convex polyhedra are a subset of presburger formulae so this method can be used to compute the number of integer points inside the polyhedra as well. In [17] a practical implementation of this adaption was presented. Choosing n as parameter, using this method will obtain the following execution bounds of the program points $e_0, e_2, e_3 = \infty; e_1 = 1$ and $e_4, e_5 = (\text{if } n \geq 0 \text{ then } n+1 \text{ else } 0)$. However, observing that the entry edge x_0 and exit edge x_3 will always be taken exactly once, we put the more accurate

$$\begin{aligned} e_2 &= \infty \\ e_0, e_1, e_3 &= 1 \\ e_4, e_5 &= \begin{cases} n+1 & \text{if } n \geq 0 \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{2}$$

Substituting the bounds $e_{i \in \mathcal{Q}_L}$ in (2) for the symbolic bounds $p_{i \in \mathcal{Q}_L}$ in (1) will result in a formula parametric in n. After manual simplification we get the following parametric WCET formula:

$$\text{PWCET}_L = \begin{cases} 30n + 70 & \text{if } n \geq 0 \\ 40 & \text{otherwise.} \end{cases}$$

IV. The Minimum Propagation Algorithm

This section describes a new parametric calculation algorithm called *Minimum Propagation Analysis* (MPA). It operates on the CFG of a program, where each edge q (program point) has a worst-case timing c_q, a maximum execution count v_q, and a symbolic capacity p_q such that

$$v_q \leq p_q \tag{3}$$

is guaranteed to hold. Then, an obvious but possibly not precise estimation of the WCET of the program is:

$$\sum_{q \in \mathcal{Q}} c_q p_q. \tag{4}$$

Furthermore, an edge can not be visited more times than the sum of its predecessors or the sum of its successors. This can be expressed formally as

$$v_q \leq \sum_{q' \in \text{pred}(q)} v_{q'} \tag{5}$$

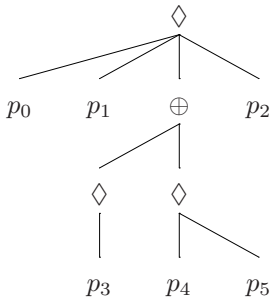
$$v_q \leq \sum_{q' \in \text{succ}(q)} v_{q'} \tag{6}$$

Now, we have three upper bounds: (3), (5) and (6), for v_q. Obviously the smallest one of these is the tightest and most desirable one. We call the smallest possible bound t_q and substitute it for p_q in (4).

MPA attempts to find the tightest possible bound for a program point using (3), (5) and (6). As can be seen, the bound of a program point depends on the bounds of its neighbours. The tightest bound of a program as a formula is therefore quite intricate. MPA models the intermediate and final bounds of program points as a tree which is explained in next section.

A. The Min-Tree

The upper bound for a program point needs to be valid for all possible combinations of symbolic execution counts $p_{q \in \mathcal{Q}}$. An upper bound t_q will be represented as a tree with three types of nodes: minimum nodes, plus nodes and leaf nodes. Minimum nodes (denoted $\Diamond$) expresses the minimum of all its children. Plus nodes (denoted $\oplus$) expresses the sum of all its children. Leaf nodes (denoted p_q) expresses the value of p_q. Such a tree will be referred to as a *Min-Tree*. Below is an example of a Min-Tree.



This Min-Tree represents the expression:

$$\min(p_0, p_1, p_3 + \min(p_4, p_5), p_2).$$

B. The Algorithm

MPA is shown in Algorithm 1. It is a recursive procedure which takes as argument a program point, a context and a set of constraints. The algorithm returns a Min-Tree as described in previous section. The context is simply a set keeping track of visited program points and is the empty set in the first call. The set of constraints corresponds to that of (5) and (6) and is obtained directly from the graph structure. The constraint (3) is implicit and is not needed in the constraint argument of the algorithm. MPA searches the given constraints and recursively builds a Min-Tree by adding visited nodes as children to a minimum node. It searches all simple paths first, leaving the branches for later. The branches are then recursively computed as children for plus nodes.

The root of the Min-Tree will always be a minimum node, and its children will be all maximum bounds found for the program point under analysis. MPA maintains a worklist and a branch set; the worklist keeps track of visited program points and the branch set keeps track of pending plus nodes. Whenever a program point has single predecessors and successors, the neighbouring capacities alone constitutes as upper bounds for the program point and is therefore put in the worklist for continued processing. In the case of branching program points, the program points are put in the pending branch set for recursive processing as children of a plus node. The reason for this can be read directly from (5) and (6), where it is obvious that it is the *sum* of the upper bounds of the other program points that needs to be computed.

Algorithm 1 MPA$(v_i, \text{context}, \text{constraints})$

```
 1: node ← mkMinNode()
 2: worklist ← push(NIL, i)
 3: branch ← ∅
 4: while worklist ≠ NIL do
 5:    k ← peek(worklist)
 6:    worklist ← pop(worklist)
 7:    if k ∉ context then
 8:       context ← context ∪ {k}
 9:       node ← addLeaf(node, p_k)
10:       for all [v_k ≤ v_j] ∈ constraints do
11:          if j ∉ context then
12:             worklist ← push(worklist, j)
13:          end if
14:       end for
15:       for all [v_k ≤ Σ_{n∈N} v_n] ∈ constraints do
16:          if N ∩ context = ∅ then
17:             branch ← branch ∪ {N}
18:          end if
19:       end for
20:    end if
21: end while
22: for all N ∈ branch do
23:    plusNode ← mkPlusNode()
24:    for all n ∈ N do
25:       child ← MPA(n, context, constraints)
26:       plusNode ← addChild(plusNode, child)
27:    end for
28:    node ← addChild(node, plusNode)
29: end for
30: return node
```

C. Detailed Explanation of the Algorithm

Row 1 creates the root of the tree which is always a minimum node, the primitive *mkMinNode* returns a minimum node without children. Row 2-3 initialises the worklist and the branch set. The work list is implemented as a stack and using the stack primitives *push*, *pop* and *peek* (peek returns the top element from the stack without removing it) to manipulate it. The loop in row 4-21 is building the leaves of the minimum node and puts the pending plus nodes in the branch set. Row 7 ensures that nodes which have already been considered (and thus don't contribute to any tighter result) are skipped. Row 9 adds leaves to the minimum node by using the primitive *addLeaf* which takes a node and a leaf and returns the node with the leaf added. Then, on row 10-14, all single entry/exit constraints are added to the worklist for further processing. Row 15-19 adds the multiple entry/exit constraints to the pending branch set.

When no more program points are present in the worklist, the algorithms enters next step (row 22). By now, *node* is a minimum node, possibly with a couple of leaves, that all constitute maximum bounds on the program point i. In other words, the constraints from (3) have been added. Left to add is the plus nodes, which correspond to (5) and (6). This is done in row 16-23. Each branching constraint will produce a plus node (row 23), this is done by the primitive *mkPlusNode* which

simply returns a plus node without children. The children of the plus node are then recursively computed from each term in the constraint (row 26), and then added as children to the plus node via the primitive *addChild* (row 26). Finally, each plus node is added as a child of the minimum node (row 28) and the root node is returned (row 30).

D. Correctness

This section will give an informal argument that MPA is correct. By correct we mean that edge q is guaranteed to be visited less than or equal to the expression represented by the Min-Tree returned by $MPA(v_q, \varnothing, \text{constraints})$. First, we observe that the leaf node added on row 9 of Algorithm 1 is correct. This leaf expresses that $v_q \leq p_r$ where r is any program point which may be in the worklist. The program points that may be in the worklist are the neighbours to q (including q itself) which *must* be or must have been visited when q is visited (see row 10). All nodes in this set of program points must be visited the same number of times, say m. Since all nodes in this set must be visited every time q is visited, the least capacity of these nodes constitutes maximum bound on m. Thus, the algorithm is correct if no branches occurs, if we by inductive hypothesis assume that MPA is correct we can also show that the recursive case is also correct. Every set of program points N in the branch set represent a selection of edges in the CFG, that is, exactly one of the program points in N will be taken for every time program point q is visited. This means that the sum of all upper bounds of the program points in N is an upper bound also on the number of times q can be visited. By inductive hypothesis, we can use MPA to obtain correct bounds for all nodes in N, and by adding the plus node (on row 26), we will obtain an additional correct upper bound of program point q.

E. Example

Consider the example program L in Figure 2. Each program point has parametric capacities $p_0, ..., p_5$. We will show how to compute an upper bound for v_0. The set of constraints obtained from (3),(5) and (6) are the following

$$\begin{aligned}\forall q \in \mathcal{Q}(v_q \leq p_q)\\ v_0 \leq v_1\\ v_1 \leq v_2, v_0\\ v_2 \leq v_1 + v_5, v_3 + v_4\\ v_3 \leq v_2\\ v_4 \leq v_2, v_5\\ v_5 \leq v_4, v_2.\end{aligned}$$

We start by calling $\text{MPA}(v_0, \varnothing, \text{constraints})$. Processing in row 4-21 will generate the following intermediate results:

$\text{analysis}(v_0, \varnothing, \text{constraints})$			
node	worklist	branch	context
min()	[0]	$\varnothing$	$\varnothing$
min(p_0)	[1]	$\varnothing$	$\{0\}$
min(p_0,p_1)	[2]	$\varnothing$	$\{0,1\}$
min(p_0,p_1,p_2)	[]	$\{\{3,4\}\}$	$\{0,1,2\}$

After the worklist has become empty and the main loop has finished, the algorithm is in row 22 and the plus nodes will be evaluated. We have that $N = \{3,4\}$ and so this leads to two recursive calls: $\text{MPA}(v_3, \{0,1,2\}, \text{constraints})$ and $\text{MPA}(v_3, \{0,1,2\}, \text{constraints})$. The following tables shows the intermediate results for these calls.

$\text{MPA}(v_3, \{0,1,2\}, \text{constraints})$			
node	worklist	branch	context
min()	[0]	$\varnothing$	$\{0,1,2\}$
min(p_3)	[]	$\varnothing$	$\{0,1,2,3\}$

$\text{MPA}(v_4, \{0,1,2\}, \text{MPA})$			
node	worklist	branch	context
min()	[4]	$\varnothing$	$\{0,1,2\}$
min(p_4)	[5]	$\varnothing$	$\{0,1,2,4\}$
min(p_4,p_5)	[]	$\varnothing$	$\{0,1,2,4,5\}$

The result of these two calls will both be children to a plus node, which in turn will be child to the minimum node that will be returned from the original call. This plus node is then added as child to the previous minimum node. The final Min-Tree for v_0 expresses:

$$\min(p_0, p_1, p_2, \text{plus}(\min(p_3), \min(p_4, p_5))) \ .$$

The tightest upper bounds for each program point $q \in \mathcal{Q}$ that was found can be found by MPA is the corresponding Min-Tree $t_{q \in \mathcal{Q}}$. We can therefore estimate the WCET of a program by

$$\text{WCET}_P = \sum_{q \in \text{Q}} c_q t_q.$$

The Min-Trees of L represent

$$\begin{aligned}t_0 &= \min(p_0, p_1, p_3 + \min(p_4, p_5), p_2)\\ t_1 &= \min(p_0, p_1, p_3 + \min(p_4, p_5), p_2)\\ t_2 &= \min(p_2, \min(p_0, p_1) + \min(p_4, p_5), p_3 + \min(p_4, p_5))\\ t_3 &= \min(p_2, p_3, \min(p_0, p_1) + \min(p_4, p_5))\\ t_4 &= \min(p_2, p_4, p_5)\\ t_5 &= \min(p_2, p_4, p_5).\end{aligned}$$

Setting $c_{q \in \mathcal{Q}} = 10$ will result in the following:

$$\begin{aligned}&\text{WCET}_L =\\ &10(2 \cdot \min(p_0, p_1, p_3 + \min(p_4, p_5), p_2)\\ &+\min(p_2, \min(p_0, p_1)\\ &+\min(p_4, p_5)), p_3 + \min(p_4, p_5)\\ &+\min(p_2, p_3, \min(p_0, p_1)\\ &+\min(p_4, p_5))\\ &+2 \cdot \min(p_2, p_4, p_5)).\end{aligned} \tag{7}$$

Substituting the parametric execution bounds $e_{i \in \mathcal{Q}_L}$ in (2) of Section III-A for the execution count parameters $p_{i \in \mathcal{Q}_L}$ in

TABLE I
TEST RESULTS

Benchmark	Pps	PIP Exec. time	KB
edn (fir)	11	0.1s	3/1
edn (latsynth)	7	0.1s	1/1
edn (latsynth)x2	12	0.1s	2/1
edn (latsynth)x4	25	0.1s	10/3
cnt (initialize)	12	0.1s	3/1
cnt (initialize)x2	23	0.3s	83/3
cnt (initialize)x3	34	5.6s	1782/6
cnt (sum)	16	0.3s	80/2
cnt (sum)x2	31	-	-/5
jcomplex	23	-	-/7
matmult (Initialize)	12	0.1s	3/1
matmult (Initialize)x2	23	0.3s	83/3

(7) will after manual simplification yield:

$$\mathrm{PWCET}_L = \begin{cases} 30n + 70 & \text{if } n \geq 0 \\ 40 & \text{otherwise.} \end{cases}$$

Which is the same we obtained from PIP, exemplified in Section III-A.

V. EVALUATION

In this section we will evaluate MPA in two ways. First, we will compare the precision, execution time and solution size to the previously suggested method PIP. This is done by running the two approaches on a prototype of the method described in Section III. This prototype was described in [17]. Since this prototype is somewhat limited in its implementation, the scalability of MPA is also evaluated by running it in isolation with full-scale benchmarks.

A. PIP

Parametric Integer (linear) Programming (PIP) was introduced in [11] and can be used as a generalisation of integer linear programming. PIP allows an integer linear problem to contain unknown constants, i.e. parameters. The parameters are present in the solution that PIP gives; the solution is a nested conditional expression in terms of the parameters. By instantiating parameters one can obtain a concrete constant solution. There exists a tool called Pip [18] implementing this algorithm. This is the tool we have used in our experiments. A sample of output from PIP is shown in Section III-A.

B. Comparison with PIP

The experiments have been performed using an enhanced version of the prototype implementation found in [17]. This prototype implements most of the functionality presented in [8]. The prototype is developed in C++ and Haskell, implementing the boxes in Figure 1 as modules which communicates through text files. It performs analysis over a simple imperative language based on C. Program variables have to be integers or Booleans. Pointer, array and non-integer variables may be present in the program, but are considered having unknown values in the analysis. All function calls are inlined before analysis so the analysis works as an intra-procedural analysis on the inlined code. The tool does not have any low-level analysis, so all program points are assumed to have a constant WCET estimate of 10 clock cycles. In practise, these values could be provided from a low-level analysis.

The times given below are the real-times obtained from the UNIX command *time*. Both algorithms are implemented in C++[1] and compiled with GCC 3.4.4 under Cygwin. The program that instantiates formulae is implemented in Haskell and compiled with the Glasgow Haskell Compiler 6.8.3 under Cygwin. The experiments are run under Windows XP Professional SP3 on an Intel core duo 2.4 GHz with 2.39GB RAM and a 6MB L2-cache.

The experiments have been performed by analysing some benchmarks using both PIP and MPA. These benchmarks have been manually translated to the simplified analysed language. After analysis, some sample points in the parameters have been chosen and instantiated. In order to only compare the calculation methods, the instantiation process is made slightly different than as outlined in Section III; rather than substituting the parametric execution bounds for the execution count parameters of the parametric calculation and simplify, the parametric execution bounds are first calculated and directly substituted into the solution of the parameteric calculation. As an example, in (1) of Section III-A, a concrete value for n is used to instantiate $a_{i \in \mathcal{Q}_L}$ of (2) to concrete values. The instantiated values of $a_{i \in \mathcal{Q}_L}$ are directly substituted for $p_{i \in \mathcal{Q}_L}$ in the formulae (1) and (7).

Thus, the time it takes to substitute parametric execution bounds for execution count parameters and simplify is not included in the calculations. When using the method in practise, less time will be consumed for instantiating (due to simplifications), and more time to perform analysis (due to simplification time).

We will now explain how the comparison was made. The columns of Table I is explained from left to right.

1) Benchmark: The benchmarks are taken from the Mälardalen benchmarks [19]. These benchmarks are common to use when evaluating WCET methods. We have chosen benchmarks that conform to the limitation of the analysis tool and which have a timing behaviour which is parametric in some variables or constant macros. The particular function which has been analysed is the name within parentheses. When a benchmark is marked with *x2,x3* etc, it means that the particular function has been called repeatedly and thus inlined multiple times. This is just to see how the analysis scales in increased number of program points.

2) Program points: Labelled as *Pps* in the table. This is the number of edges in the control flow graph.

3) Execution time: The execution time given is for Pip, since MPA solves all these benchmarks in less than 0.2 seconds. The cases where the time is not given means that Pip couldn't solve the problem due to a too high complexity of the solution.

4) Solution size: The solution sizes are given in KB, first is Pip, second is MPA. The measurements comes from the file sizes of the solutions textual representations.

[1]Pip is open source

TABLE II
PRECISION COMPARISON

Benchmark	Parameters	Pip result	MPA result	Diff	Percent
edn (fir)	N = 100, ORDER = 25	60790	60810	20	0.03%
	N = 100, ORDER = 50	78040	78060	20	0.03%
	N = 100, ORDER = 75	57790	57810	20	0.03%
	N = 200, ORDER = 25	141790	141810	20	0.01%
	N = 200, ORDER = 50	234040	234060	20	<0.01%
	N = 200, ORDER = 75	288790	288810	20	<0.01%
	N = 300, ORDER = 25	222790	222810	20	<0.01%
	N = 300, ORDER = 50	390040	390060	20	<0.01%
	N = 300, ORDER = 75	519790	519810	20	<0.01%
edn (latsynth)	n = 50	1520	1520	0	0%
	n =100	3020	3020	0	0%
	n =200	6020	6020	0	0%
edn(latsynth)x2	n = 50	3030	3060	30	0.99%
	n = 100	6030	6060	30	0.5%
	n = 200	12030	12060	30	0.25%
edn(latsynth)x4	n = 50	6050	6160	110	1.82%
	n =100	12050	12160	110	0.91%
	n =200	24050	24160	110	0.46%
cnt (initialize)	MAXSIZE=10	4640	4660	20	0.4%
	MAXSIZE=20	17240	17260	20	0.1%
	MAXSIZE=30	37840	37860	20	0.05%
cnt (initialize)x2	MAXSIZE=10	9270	9810	540	5.83%
	MAXSIZE=20	34470	35510	1040	3.02%
	MAXSIZE=30	75670	77210	1540	2.04%
cnt (initialize)x3	MAXSIZE=10	13900	15460	1560	11.22%
	MAXSIZE=20	51700	54760	3060	5.92 %
	MAXSIZE=30	113500	118060	4560	4.018%
cnt (sum)	MAXSIZE=10	6640	8660	2020	30.4%
	MAXSIZE=20	25240	33260	8020	31.8%
	MAXSIZE=30	55840	73860	18020	32.3%
cnt(sum)x2	MAXSIZE=10	-	17810	-	-
	MAXSIZE=20	-	67510	-	-
	MAXSIZE=30	-	149210	-	-
jcomplex	a = 1, b = 1	-	80	-	-
	a = 1, b = 15	-	120	-	-
	a = 1, b = 30	-	110	-	-
	a = 15, b = 1	-	80	-	-
	a = 15, b = 15	-	80	-	-
	a = 15, b = 30	-	30	-	-
	a = 30, b = 1	-	80	-	-
	a = 30, b = 15	-	80	-	-
	a = 30, b = 30	-	30	-	-
matmult (Initialize)	UPPERLIMIT = 100	406040	406060	20	<0.01%
	UPPERLIMIT = 150	909040	909060	20	<0.01%
	UPPERLIMIT = 200	1612040	1612060	20	<0.01%
matmult (Initialize)x2	UPPERLIMIT = 100	812070	817110	5040	0.62%
	UPPERLIMIT = 150	1818070	1825610	7540	0.41%
	UPPERLIMIT = 200	3224070	3234110	10040	0.31%

Note that PIP does not scale well, especially not in solution sizes. PIP sometimes fails to produce a solution, even for quite small programs.

Table II shows the estimated WCETs when the chosen parameters have been instantiated in the parametric formulae from the two parametric calculation methods. The parameters have been chosen so they have a parametric behaviour and are instantiated with values somewhat close to their original values in the benchmark programs. The last two columns shows how much the MPA solution differs from the PIP solution in that particular instantiation. As can be seen, MPA gives slightly less precise result compared to PIP. As high imprecision 32.3% has been observed, but in most cases it is less than one percent.

C. Scaling Properties

Since the translated benchmarks used in previous section are small, they don't show the scaling properties of MPA properly. In order to investigate how MPA scales in more realistic cases, we have run the algorithm alone (independent of the rest of the prototype) on the full benchmark suite of [19]. MPA takes only a CFG as argument, and returns a result parametric in the capacities of the graph. We have used the WCET analysis research prototype SWEET [20] to obtain CFGs for the benchmarks. Since we do not run the whole method described in Section III, we cannot examine the precision of MPA in this test, just how efficient it is.

The CFGs obtained from SWEET are on the *full programs*,

TABLE III
SCALABILITY PROPERTIES OF MPA

Benchmark	pp	mvd	Iterations	calls	time	filesize
adpcm	884	4	-	-	-	-
bs	39	2	249	1538	0.04s	6633
bsort	66	2	1750	9610	0.14s	44152
cnt	93	2	1537	11547	0.16s	48022
cover	1593	121	-	-	-	-
compress	380	5	-	-	-	-
crc	127	2	10543	69017	0.85s	316212
duff	390	9	5937	121369	1.22s	475556
edn	342	2	7202	95585	1.04s	421438
expint	88	2	1028	7983	0.11s	32407
fac	36	2	260	1298	0.08s	5959
fdct	147	2	973	21565	0.22s	79457
fft1	266	4	72572	482486	6.18s	2390541
fibcall	29	2	75	702	0.03s	2567
fir	77	2	779	6828	0.10s	27315
insertsort	39	2	175	1313	0.03s	5310
jcomplex	48	2	792	3289	0.06s	16763
jfdctint	122	2	1038	14726	0.16s	55563
lcdnum	158	17	4042	40164	0.46s	168927
lms	262	4	90864	586176	7.44s	2959463
ludcmp	181	3	5583	35101	0.47s	169103
matmult	97	2	1351	9441	0.13s	40263
minmax	109	3	1926	18881	0.23s	74619
ndes	445	9	1235359	11593218	2m19s	54938649
ns	46	2	562	2838	0.06s	13245
nsichneu	3313	5	-	-	-	-
prime	114	3	11425	79060	0.95s	356992
qsort-exam	153	2	15861	104870	1.30s	501762
qurt	135	4	27658	178578	2.17s	821808
select	136	2	32418	165320	2.25s	842275
sqrt	49	3	896	4717	0.08s	21758
statemate	1287	47	-	-	-	-
ud	150	2	2770	15938	0.23s	78277

that is, it includes all functions and all function calls. In contrast to the evaluation in Section V, the CFGs obtained from SWEET are not inlined; each function call is an edge from the caller to the callee, and each return is an edge from the exit of a function, back to the caller.

Table III shows the result of the tests. The first column is the benchmark name, second column is the number of program points. The third column is the maximum degree of a node, i.e. the maximum number of emerging or incoming edges from a node. The fourth columnn is the global number of iterations of MPAs main loop (row 4-21 in Algorithm 1). The fifth column is the global number of calls (including recursive calls) to Algorithm 1. The sixth column is the real time of the algorithm running, obtained by the UNIX command *time*. Finally, the seventh column is the size of the solution file in bytes.

Five of the programs fails to be analysed, due to memory failure. The reason seems to be the combination of many program points and a high vertex degree on the nodes, resulting in a high number of recursive branches. However, most of the programs can be analysed, and with a good efficiency.

VI. RELATED WORK

This section presents related work in the field of parametric WCET. In [21], a WCET analysis which computes a formula given in some chosen set of function parameters, is presented. Flow constraints has to be manually provided. Two methods of parametric WCET analyses are presented in [22] and Coffman [23]. They are both parameterised in loop bounds only, they do not take global constraints into consideration. A method similar to Lisper's were presented in [24], but it is using loop and path analyses instead of abstract interpretation. It requires special treatment of loops and is not as accurate as polyhedral abstract interpretation. A method which computes complexity of a program is presented in [25]. This method derives symbolic bounds of the complexity of the code only and does not take hardware into consideration, and cannot be used to obtain WCET estimations. A prototype implementation of Lisper's method was presented in [17] which is the implementation on which we have done our experiments.

VII. SUMMARY AND CONCLUSION

We have presented a new algorithm called Minimum Propagation Analysis (MPA) which performs parametric calculation on programs. The algorithm is significantly more efficient both in speed and memory compared to the previously suggested method PIP, to the cost of some imprecision.

With this efficient algorithm we have shown that a parametric WCET analysis can be performed on somewhat realistic program examples without too much problems. We have also shown that a previously suggested method (PIP), scales badly on larger programs and therefore needs to be replaced by a better algorithm.

The algorithm is designed to be used for the method outlined in [8], but can be useful even in other applications. The method computes a WCET formula in terms of symbolic upper bounds on program points; the concrete values on these bounds could be instantiated in any (safe) way, and is not restricted to be used only in the method outlined in [8]. In general, MPA could potentially be useful in other applications than WCET analysis since it operates on general graphs with parametric capacities.

VIII. Future work

Future plans are to improve MPA to be able handle the cases where it currently fails. Ideas include implementing more efficient data types for representing the analysis values, saving intermediate results to disk or eliminating the recursion. Other plans are to find exact complexity bounds of MPA as well as investigating the source of over-approximation.

A full implementation of the method in [8] in the static WCET analysis tool SWEET [20] is planned for further investigation and a full evaluation of the method.

References

[1] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper, "Applying static WCET analysis to automotive communication software," in *Proc. 17^{th} Euromicro Conference of Real-Time Systems, (ECRTS'05)*, Jul. 2005.

[2] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper, "Static timing analysis of real-time operating system code," in *Proc. 1^{st} International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, Oct. 2004.

[3] M. Carlsson, J. Engblom, A. Ermedahl, J. Lindblad, and B. Lisper, "Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system," in *Proc. 2^{nd} International Workshop on Real-Time Tools*, 2002.

[4] I. Crnkovic, "Component-based software engineering for embedded systems," in *International Conference on Software engineering, ICSE'05*. ACM, 5 2005.

[5] G. T. Heineman and W. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Professional, June 2001.

[6] H. Hansson, M. Akerholm, I. Crnkovic, and M. Torngren, "Saveccm - a component model for safety-critical real-time systems," in *EUROMICRO '04: Proceedings of the 30th EUROMICRO Conference*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 627–635.

[7] "Arcticus Systems homepage," 2009, `www.arcticus-systems.com`.

[8] B. Lisper, "Fully automatic, parametric worst-case execution time analysis," in *Proc. 3^{rd} International Workshop on Worst-Case Execution Time Analysis (WCET'2003)*, J. Gustafsson, Ed., Porto, Jul. 2003, pp. 77–80.

[9] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution time problem — overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[10] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'95)*, La Jolla, CA, Jun. 1995.

[11] P. Feautrier, "Parametric integer programming," *Operationnelle/Operations Research*, vol. 22, no. 3, pp. 243–268, 1988. [Online]. Available: citeseer.ist.psu.edu/feautrier88parametric.html

[12] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4^{th} ACM Symposium on Principles of Programming Languages*, Los Angeles, Jan. 1977, pp. 238–252.

[13] A. Miné, "The octagon abstract domain," *Higher Order Symbol. Comput.*, vol. 19, no. 1, pp. 31–100, 2006.

[14] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proc. 5th ACM Symposium on Principles of Programming Languages*, 1978, pp. 84–97.

[15] P. Granger, "Static Analysis of Arithmetical Congruences," *International Journal of Computer Mathematics*, pp. 165–199, 1989.

[16] W. Pugh, "Counting solutions to Presburger formulas: How and why," in *SIGPLAN Conference on Programming Language Design and Implementation*, 1994, pp. 121–134. [Online]. Available: citeseer.ist.psu.edu/pugh94counting.html

[17] S. Bygde and B. Lisper, "Towards an automatic parametric WCET analysis," in *Proc. 8^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2008)*, R. Kirner, Ed., Prague, Czech Republic, Jul. 2008, pp. 9–17.

[18] "Piplib website," 2009, `http://www.piplib.org/`.

[19] "Mälardalen WCET benchmarks homepage," 2009, `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[20] A. Ermedahl and J. Gustafsson, "Deriving Annotations for Tight Calculation of Execution Time," in *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*. Springer Verlag, Aug 1997, pp. 1298–1307.

[21] A. Colin and G. Bernat, "Scope-tree: a program representation for symbolic worst-case execution time analysis," in *Proc. 14^{th} Euromicro Conference of Real-Time Systems, (ECRTS'02)*, Vienna, Jun. 2002, pp. 50–59.

[22] E. Vivancos, C. Healy, F. Mueller, and D. Whalley, "Parametric timing analysis," in *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'01)*, J. Fenwick and C. Norris, Eds., Snowbird, Utah, Jun. 2001, pp. 88–93.

[23] J. Coffman, C. Healy, F. Mueller, and D. Whalley, "Generalizing parametric timing analysis," in *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools*. New York, NY, USA: ACM, 2007, pp. 152–154.

[24] S. Altmeyer, C. Hümbert, B. Lisper, and R. Wilhelm, "Parametric timing analysis for complex architectures," in *Proc. 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 08)*, Kaohsiung, Taiwan, Aug. 2008.

[25] S. Gulwani, K. K. Mehra, and T. Chilimbi, "Speed: precise and efficient static estimation of program computational complexity," in *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2009, pp. 127–139.