A Symbolic Execution Method for Bounds Analysis

Joxan Jaffar, Andrew E. Santosa, D.H. Chu and Jorge Navas

Department of Computer Science
National University of Singapore
Republic of Singapore 117543
{joxan,andrews,chuduchi,navas}@comp.nus.edu.sg

Abstract. Given a program whose loops are bounded, we address the problem of estimating the upper bound of a variable which is monotonically increasing, and its typical application in annotating a program so that bounds analysis produces an estimate of the worst-case resource usage. The method presented is a systematic enumeration of symbolic states of the program. The novelty is twofold: first, we use *intermittent invariants*, each being a property that is true of some but not all iterations of a loop. This allows for the discrimination of analyses of different iterations while still providing an abstraction so that the analysis can be practical. Second, we compute the bound estimate by simulating the behavior of the loop using a *dynamic programming* algorithm, relieving us from discovering a closed form expression for the loop. The analysis time of a loop, which reasons about *all* inputs, is then proportional to the actual running time of the loop, on *one* input. The method compared formally with prior work, and evaluated empirically on benchmark programs.

1 Introduction

Predicting upper bounds on the resource usage is needed to verify that constraints are not violated in hard real-time systems. The most important resource is often the execution time of a program. Unfortunately, the problem of inferring upper bounds on the execution time of a program is undecidable. Instead, some restrictions are forced to guarantee the program to be analyzed always terminates. Even so, to estimate the execution time is still of huge complexity [22]. Since all possible input value combinations must be considered, the number of possible program paths increases exponentially with the number of control flow branches. Therefore, testing all executable paths is unfeasible when dealing with larger applications.

Worst-Case Execution Time (WCET) analysis aims to provide the worst possible execution time of a program used in a system and it is usually performed at two different levels [18]. The *low-level*, which is done at the object code, provides the execution time of basic blocks considering the effects of hardware level features such as cache, pipelining, branch prediction, etc. [9, 5]. On the other hand, the *high-level* analysis is performed at the source code and it focuses on characterizing possible execution paths. A big advantage of performing the high level analysis in a separate step is that the hardware model can be replaced independently of the rest of the tool, making a tool much easier to retarget [23].

A main issue in WCET analysis is to avoid pessimism (overestimation) in timing evaluation by providing *tight* bounds. One part of the overestimation is due to the presence of some hardware features that affect the execution time of instructions. On the other hand, to achieve a tight estimation we need information about the program behavior such as infeasible paths and maximum number of loop iterations [4] which are often provided by users. The design of an accurate and practical WCET analysis should consider at least the following features:

1. **Input Data dependent**: the WCET of a piece of code depends on the value of its input variables, where we would like to consider all possible executions efficiently, for example, in the following program, where `a` is an input array:

   ```
   if (a[i][j] != 0) {...} else {...}
   ```

2. **Non-rectangular loops**: the maximum number of iterations of an inner loop depends on the iteration number of an outer loop. For example:

   ```
   for (i=0; i < N ; i++){ for (j=0; j < i; j++) {...} }
   ```

   Here we would like to express the WCET of the inner loop as a function of some variables (e.g., `i`) defined in the scope of the outer loop.
3. **Mutually exclusive paths**: the execution of a piece of code forbids the execution of another piece of code. The following example [2], we want to take into account that the paths (a,c) and (b,d) are mutually exclusive:

   ```
   if (E < 0){ condition = 0; x = E + 45;} /* a */
   else{ condition = 1; x = -E;}           /* b */
   if (condition) result = x / y;          /* c */
   else result = y;                        /* d */
   ```

4. **Down-sampling code**: the execution of one part of the body of a loop is executed less often than the rest of the body. For example, in the following code we would like to consider the WCET of the inner if-statement only once.

   ```
   for (i=0 ; i < N; i++){ if (i == 4) {...} }
   ```

5. **Closed-form is not always possible**: the WCET analysis can produce symbolic expressions which are solved (closed-form) by using off-the-shelf *Computational Algebraic Systems* (*CAS*). However, to obtain a closed-form can be unrealistic [24]. As an example, consider the well-known Collatz problem (also called $3x+1$ problem) written as a program, which total correctness is yet to be proven (see e.g., [1] and its references):

   ```
   for( ; n != 1; ) { if ( n % 2 == 0) n = n/2; else n = 3*n + 1; }
   ```

Furthermore, WCET estimates must be also *safe*, i.e., guaranteed not to underestimate the execution time, in order to be valid for use in hard real-time systems. Therefore, the correctness depends not only on the results of the computations, but also on the time at which the result is provided. To be able to guarantee the correctness, we also need a proof of the WCET estimate [4, 7, 16].

1.1 Summary of Contributions

In this paper, we present a general method for inferring and proving tight bounds on the resource usage of programs in which termination is guaranteed, and applied this result to the high-level analysis of Worst-Case Execution Time (WCET). We start with a universally used basic methodology of not unrolling loops, but to treat a loop in a sequence of statements as just one statement to be analyzed. In order to be safe, the loop body must be analyzed not with the current context, but with a loop invariant, obtained by a simple invariant discovery method applied to the current context.

The first innovation is to allow specializations of this loop invariants in *intermittent invariants*, i.e. each one is true for some but not all iterations of a loop. This allows for the discrimination of different cases within a loop improving the accuracy of the overall method but still being practical. Since the automatic discovery of these invariants is in general infeasible, our method depends on manual intervention for this purpose. Even so, we believe the methodology is intuitive and easy to use, given that the programmer knows about the essential properties that give rise to different bounds in the loop body.

The second innovation is, in contrast to the general approach of analyzing a loop by only analyzing its body, is in fact to unroll the loop and discover the bound. This is performed at a different stage in the analysis, at the time when the analysis of its body *has been completed*. What is crucial here is that this unrolling and optimization problem is amenable to *dynamic programing*. Thus the cost is proportional to the actual running time of the loop and the number of intermittent invariants employed. In contrast, all prior methods need to discover a closed form expression for the loop, which is not just impossible in principle, but also impractical.

We finally show, using standard benchmarks, that our method not only runs in good time, but produces accurate (and often *exact*) results.

1.2 Related Work

High level WCET analysis has been the subject of much research, and substantial progress has been made in the area (see [18, 25] for surveys of WCET). There are three main categories of high level analysis methods proposed in literature: *tree-based*, *path-based*, and *IPET* (*Implicit Path Enumeration Technique*).

- In tree-based approach [19, 17, 16, 2, 5], the WCET is generated by a bottom-up traversal of a tree, generally corresponding to the control flow graph or syntax tree of the program, using rules defined for each type of compound program statement (if-statements, function calls, loops, etc).
- In path-based approach [3, 9, 22, 23], the WCET estimate is generated by calculating times for different paths in a program and searching for the path with the longest execution time. The key feature is that possible execution paths are explicitly represented which may be valuable information for the programmer, e.g. for debugging purposes.
- In IPET [13, 20, 15, 6, 8], the program flow is modeled using arithmetic constraints. Each basic block and program flow edge in the program is given a time variable (t_{entity}) and a count variable (x_{entity}), and the goal is to maximize the $\sum_{i \in entities} x_i * t_i$, subject to constraints expressing the structure of the program and possible flows. The result is the worst-case count for each node and edge.

The tree and path-based approaches have problems with flow information stretching across loop-nesting levels [8]. Moreover, tree-based methods cannot capture dependencies across statements since the computations are local within a single program statement [23]. In purely IPET methods the computational results are not better than the results of a tree-based method [2]. However, complex flows can be expressed using user-definable constraints [6, 8] but the computational complexity of solving the resulting problem is potentially exponential, since the program is completely unrolled and all flow information is lifted to a global level. Moreover, the correctness of those constraints is not verified and the WCET may be untight or, worse, unsafe.

To the best of our knowledge our method is the first bounds analysis that can prove upper bounds of the longest executable path of a program in the presence of complex features such as input data dependencies, non-rectangular loops, down-sampling code, and mutually-exclusive paths even when a closed-form can not be obtained by traditional CAS.

2 Preliminaries

We will use the formalism of Constraint Logic Programming (CLP) [10] in this paper.

We consider integer and array terms. Integer terms are constructed in the usual way, with one addition: the array element. The latter is defined recursively to be of the form $a[i]$ where a is an *array expression* and i an integer term. An array expression is either an array variable or of the form $\langle a, i, j \rangle$ where a is an array expression and i, j are integer terms. A *constraint* is either an integer equality or inequality, an equation between array expressions. The meaning of a constraint is defined in the obvious way.

In what follows, we use constraint to mean either an atomic constraint or a conjunction of constraints. We shall use the symbol ψ or Ψ, with or without subscripts, to denote a constraint.

An *atom* is of the form $p(\tilde{t})$ where p is a user-defined predicate symbol and $\tilde{t}$ a tuple of terms, written in the language of an underlying constraint solver. A *rule* is of the form $A\text{:-}\Psi, \tilde{B}$ where the atom A is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and constraint Ψ constitute the *body* of the rule. The constraint Ψ is also written in the language of the underlying constraint solver, which is assumed to be able to decide (at least reasonably frequently) whether Ψ is satisfiable or not. In our examples, we assume an integer and array constraint solver, as described below.

A *program* is a finite set of rules. A *goal* has exactly the same format as the body of a rule.

A *substitution* θ simultaneously replaces each variable in a term or constraint e into some expression, and we write $e\theta$ to denote the result. A *renaming* is a substitution which maps each variable in the expression into a distinct variable. A *grounding* is a substitution which maps each integer or array variable into its intended universe of discourse: an integer or an array. Where Ψ is a constraint, a grounding of Ψ results in *true* or *false* in the usual way.

A *grounding* θ of an atom $p(\tilde{t})$ is an object of the form $p(\tilde{t}\theta)$ having no variables. A grounding of a goal $\mathcal{G} \equiv (p(\tilde{t}), \Psi)$ is a grounding θ of $p(\tilde{t})$ where $\Psi\theta$ is *true*. We write $[\![\mathcal{G}]\!]$ to denote the set of groundings of $\mathcal{G}$. We say that a goal $\mathcal{G}$ *subsumes* another goal $\mathcal{G}'$ if $[\![\mathcal{G}]\!] \supseteq [\![\mathcal{G}']\!]$.

Let $\mathcal{G} \equiv (B_1, \cdots, B_n, \Psi)$ and P denote a non-final goal and program respectively. Let $R \equiv A\text{:-}\Psi_1, C_1, \cdots, C_m$ denote a rule in P, written so that none of its variables appear in $\mathcal{G}$. Let the equation $A = B$ be shorthand for the pairwise equation of the corresponding arguments of A and B. A *reduct* of $\mathcal{G}$ using a rule R, denoted *reduct*$(\mathcal{G}, R)$, is of the form

$(B_1, \cdots, B_{i-1}, C_1, \cdots, C_m, B_{i+1}, \cdots, B_n, B_i = A, \Psi, \Psi_1)$

provided the constraint $B_i = A \wedge \Psi \wedge \Psi_1$ is satisfiable.

A *derivation sequence* for a goal $\mathcal{G}_0$ is a possibly infinite sequence of goals $\mathcal{G}_0, \mathcal{G}_1, \cdots$ where $\mathcal{G}_i, i > 0$ is a reduct of $\mathcal{G}_{i-1}$. If the last goal $\mathcal{G}_n$ is a final goal, we say that the derivation is *successful*. A *derivation tree* for a goal is defined in the obvious way.

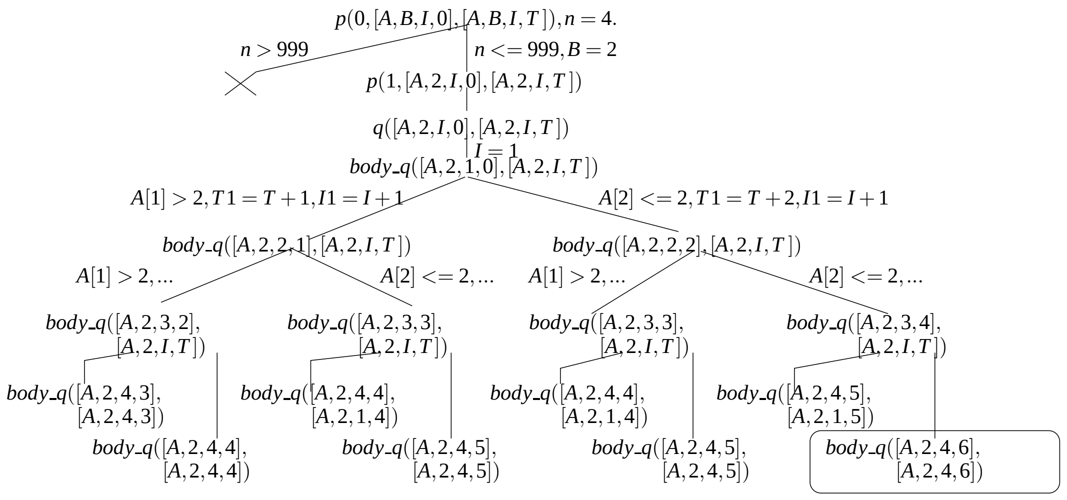
$p(0,[A,B,I,0],[A,B,I,T]), n = 4.$
$n > 999$
$n <= 999, B = 2$
$p(1,[A,2,I,0],[A,2,I,T])$
$q([A,2,I,0],[A,2,I,T])$
$I = 1$
$body\_q([A,2,1,0],[A,2,I,T])$
$A[1] > 2, T1 = T+1, I1 = I+1$
$A[2] <= 2, T1 = T+2, I1 = I+1$
$body\_q([A,2,2,1],[A,2,I,T])$
$body\_q([A,2,2,2],[A,2,I,T])$
$A[1] > 2, \ldots$
$A[2] <= 2, \ldots$
$A[1] > 2, \ldots$
$A[2] <= 2, \ldots$
$body\_q([A,2,3,2], [A,2,I,T])$
$body\_q([A,2,3,3], [A,2,I,T])$
$body\_q([A,2,3,3], [A,2,I,T])$
$body\_q([A,2,3,4], [A,2,I,T])$
$body\_q([A,2,4,3], [A,2,4,3])$
$body\_q([A,2,4,4], [A,2,1,4])$
$body\_q([A,2,4,4], [A,2,1,4])$
$body\_q([A,2,4,5], [A,2,1,5])$
$body\_q([A,2,4,4], [A,2,4,4])$
$body\_q([A,2,4,5], [A,2,4,5])$
$body\_q([A,2,4,5], [A,2,4,5])$
$body\_q([A,2,4,6], [A,2,4,6])$

Fig. 1. Execution tree for `?- p(0, [_,_,_,0], [_,_,_,T]), n = 4`

Definition 1 (Abstract Computation Tree). *An* abstract computation tree *is defined just like a derivation tree except that a node, representing a goal $\mathcal{G}$, may be replaced by another node representing another goal $\mathcal{G}'$ where $\mathcal{G}'$ subsumes $\mathcal{G}$.* □

Our concern in this paper is essentially to compute an abstract computation tree which represents all the concrete traces of the underlying program being modeled. Estimating the bound of one distinguished variable can then performed by traversing the tree.

2.1 Representing Programs as CLP Programs

In general, we shall represent a program as a transition system, expressed formally in the CLP notation. This process is straightforward, and so we will omit the details of the representation in general. Instead, we will use an example:

```
⟨0⟩ if (n > 999) b = 1 else b = 2
⟨1⟩ for (i = 1; i < n; i++) if (a[i] > b) t = t + 1 else t = t + 2
```

where the variable of interest is t. Note that for this special variable, the *resource variable*, the only operation allowed upon it is an increment of a constant, and further, the variable is not used in any other way. The CLP representation is:

```
p(0, [A,B,I,T], V) :- n > 999, B1 = 1, p(1, [A,B1,I,T], V).
p(0, [A,B,I,T], V) :- n <= 999, B1 = 2, p(1, [A,B1,I,T], V).
p(1, [A,B,I,T], V) :- q([A,B,I,T], V).
q([A,B,I,T], V) :- I1 = 1, body_q([A,B,I1,T], V).
body_q(V, V) :- I >= n.
body_q([A,B,I,T], V) :- A[I] > B, T1 = T+1, I1 = I+1, body_q([A,B,I1,T1], V)
body_q([A,B,I,T], V) :- A[I] <= B, T1 = T+2, I1 = I+1, body_q([A,B,I1,T1], V)
```

where *n* is a constant. Note that the predicate *p* represents transitions at one *level*, in this case 2, while the predicate *q*, which represents the loop, is at level 1. The first argument of *p* and *q* represent the program point, the second a list of variables representing the program state before the transition, and the third is also a list of variables but it represents the program state at *termination*. The constraints `T1 = T+1` and `T1 = T+2` shall be called the *resource constraints*.

We call a rule for a predicate of level $m+1$ which involves a predicate of level *m* (such as `p(1, [A,B,I,T], V) :- q([A,B,I,T], V)` above) a *composite* rule. All other rules are *basic*.

We call predicates such as `p` *straightline predicates*, for they describe the transitions between a sequence of program statements, and we call predicates such as `q` *loop predicates*. It is important to note that in a derivation sequence, a rule that defines a straightline predicate can be employed *at most once*.

We finally display in Fig. 1 a computation tree for this example, for $n = 4$. Note that the tree size is exponential in *n*, and clearly the bound for the resource variable *T* is 6 in the case $n = 4$. Therefore, a naive traversal of this tree, i.e., unrolling the loop, is clearly impractical. Note also that intuitively, we could have determined this answer in three steps by considering only the transitions that increment *T* by 2.

3 Analysis of Straight-Lines

The main focus of this paper is method for analyzing loops. However, it requires an algorithm for determining the WCET of straight-line programs. In fact, the loop method is *intertwined* with the straight-line algorithm.

We first assume the existence of a basic straight-line algorithm which can work on a transition system of level 1, i.e., there are no loops. Call this algorithm $A(1)$. For example, we could employ an abstract interpreter in order to get a (hopefully small) closed tree, and simply read off the maximum bound from the tree. Such a basic algorithm, in conjunction with our method for loops in the next section, induces a general algorithm $A(m)$ for straight-lines, as follows.

Suppose the underlying program is represented by a transition system of level $m+1$ for some $m \geq 1$. We now process the transitions at level $m+1$ in a sequence according the basic algorithm $A(1)$. Upon encountering a composite rule

```
p(K, V, Vf) :- q(0, V, V1), p(K1, V1, Vf).
```

where the predicate *q* is at level *m*, we now apply the loop algorithm in the next section in order to obtain a CLP program for the predicate `loop`. Note that the loop algorithm is being called for *this particular* encounter of the loop transition. We now replace the above rule with

```
p(K, V, Vf) :- loop(V, V1), p(K1, V1, Vf).
```

If we now can replace the predicate `loop(V, V1)` with a constraint, we have a transition system at only one level (with the only predicate `p`). Algorithm $A(1)$ can now be applied on the final transition system. We explain this process of replacement in more detail in Section 5.

In what follows, let the procedure *gen_strline(i, Ψ, S, Φ)* produce the CLP program `strline_i/2`[1] as described below.

[1] The notation `p/2` emphasizes that the predicate `p` takes 2 arguments.

Input: a “name” i, a start context Ψ, an end context Φ and straight-line program S

Output:

- if $\{\Psi\}S\{\Phi\}$ holds, then a CLP program `strline_i(V, Vf)` which correctly returns the WCET time $T \in V_f$ when the goal ?- Ψ, `strline_i(V, Vf)` is run. (The value T is generally *functionally dependent* on Ψ.)
- otherwise, no CLP program is generated.

In summary for this section, we assume an overall straight-line algorithm scheme for determining the bound. Though this algorithm is not the focus of this paper, it is important for accuracy because it has the potential of determining which paths through the computation tree corresponding to a straightline are possible. In our examples, we shall use a naive algorithm which generates the full computation tree (and we pick the “longest path” from here). In practice, a more advanced algorithm may be needed. An example is the interpolation-based algorithm described in [12].

4 Analysis of a Loop

Recall that to analyze a loop by extrapolating an analysis of its body requires that the body analysis takes place in the context of a loop invariant. The key idea here is that we use, instead, a (small) number of *intermittent invariants*, each of which is a true property of the state just before *some* but not *all* invocations of the loop body.

The definition of intermittent invariants is essentially manual, because it serves to partition the loop iterations into important cases. Formally, we use a collection of formulas (whose free variables are the program variables) $\mathcal{A}_1, \cdots, \mathcal{A}_m$, for some $m \geq 1$, to represent these invariants.

Let L stand for the loop

for `(init(Xs); step(Xs); exit(Xs)) S`

The procedure *gen_loop(*Ψ*, L)* generates a CLP program `loop_i/2` as described below. Before presenting it, we require a few definitions:

- `step(V, V1)`:
  is an abstraction of the transition relation of S. This is to represent that part of the loop body which executes a well-founded order in order to ensure loop termination. A typical step is the increment of a counter variable where the exit condition is that the counter exceeds a given bound. Note that, in general, only a small part of the body is needed to be considered in order to be assured of termination.
- initial context is Ψ:
  Clearly we cannot analyze the loop body under an arbitrary context such as Ψ. Instead, we require a *loop invariant*. Toward this end, define $\overline{\Psi}$ to be a generalization of Ψ so that $\{\overline{\Psi}\}S\{\overline{\Psi}\}$ holds. There is a crucial exception: $\overline{\Psi}$ *retains the resource constraints*. There is a simple implementation of this generalization: simply delete from Ψ each individual constraint that may be changed inside the loop body. (This process is therefore a form of *lightweight* invariant discovery.)
- *intermittent invariants*:
  The idea here is to allow discrimination into a small number of cases of abstract states $\mathcal{A}_1, \cdots, \mathcal{A}_m$ that precede the loop body. We call these states intermittent invariants because they are invariant only some of the time, and as such, they can

be thought as partitions of a (true) loop invariant. let $\mathcal{A}_i : i \in \{1, \cdots, m\}$ denote the subset of intermittent invariants that are entailed by Ψ. Note that there must be at least one such $\mathcal{A}_i$. Suppose the set of such i is Σ.

We can now present the procedure *gen_loop(Ψ, L)*. Note that though we use the specific predicate names `loop` and `body` below, *fresh* names are used each time the procedure is invoked.

```
write rule: loop(V, Vf) :- init(V, V1), bodyloop(V1, Vf).
write rule: bodyloop(V, V) :- exit(V).
let memo table be empty
for each $i \in \Sigma$ run gen_loop($i$)

gen_loop($i$) {
    memo $i$
    for each ($1 \leq j \leq m$) {
        gen_strline($j$, $\overline{\Psi} \wedge \mathcal{A}_i$, S, $\overline{\Psi} \wedge \mathcal{A}_j$)
        write rule:
            body(V, Vf) :-
                $\mathcal{A}_i$(V),
                strline_j(V, V1),
                step(V1, V2),
                body(V2, Vf)
        if ($j$ is not memoed) gen_loop($j$) }}
```

Note that above we have, for simplicity, considered all cases $1 \leq j \leq m$ for each case i. Clearly we only need to consider those j for which the straightline code from the abstraction $\mathcal{A}_i$ to $\mathcal{A}_j$ is possible (and this is tested in procedure *gen_strline*).

Consider the example program S

```
k = 0;
for (i = 1; i <= n; i++)
    for (j = 1; j <= i; j++)
        if (k == 0)
            if (j % 2 = 0) t++;
```

Calling the procedure *gen_strline(4,true,S,true)* on this results in:

```
strline4([I,J,K,T], Vf) :- K = 0, strline3([I,J,K,T], Vf).
strline3(V, Vf) :- loop2(V, Vf).

loop2(V,V1) :- V2=[I1,J,K,T], I1 = 1, body2(V2,V1).
body2([I,J,K,T], [I,J,K,T]) :- I > n.
body2([I,J,K,T], Vf) :-
    strline2([I,J,K,T], [I1,J1,K1,T1]),
    I2 = I1 + 1,
    body2([I2,J1,K1,T1], Vf).

strline2(V, Vf) :- loop1(V, Vf).
```

```
loop1(V,V1) :- V2=[I,J1,K,T], J1 = 1, body1(V2,V1).
body1([I,J,K,T], [I,J,K,T]) :- J > I.
body1([I,J,K,T], Vf) :-
    J = 2*Z,
    strline1even([I,J,K,T], [I1,J1,K1,T1]),
    J2 = J1 + 1,
    body1([I1,J2,K1,T1], Vf).
body1([I,J,K,T], Vf) :-
    J = 2*Z + 1,
    strline1odd([I,J,K,T], [I1,J1,K1,T1]),
    J2 = J1 + 1,
    body1([I1,J2,K1,T1], Vf).

strline1even([I,J,K,T], [I,J,K,T1]) :- K = 0, J = 2*Z, T1 = T + 1.
strline1odd(V, V).
```

A first remark is that we have used two intermittent invariants corresponding to the cases where j is even or odd. Note that we implement this test above as $J = 2*Z$ where Z is an existential variable, i.e., it does not appear in the head of the rule.

We thus get two versions of the predicate `body1`, each invoking a different version (`strline1even` and `strline1odd`) of the innermost (if) statement. Note that the creation of these versions both assumed the context $K = 0$ because this constraint was *invariant* in both loops. Further, each version had a context which stated whether J was even, and thus accurately produced an increment for T.

For this particular example, we in fact obtain, by running the initial goal

```
?- strline4([_,_,_,0], [_,_,_,Tf])
```

the *exact* bound `Tf` for any given positive constant n.

5 Executing the Compiled Program with Dynamic Programming

We now describe how we run the CLP program obtained from the compilation of an underlying program. Recall that a program, say its nesting level is m, is compiled into CLP program whose rules are of the two forms:

```
p(i, V, Vf) :- Ψ(V, V1), p(j, V1, Vf).
p(i, V, Vf) :- loop(V, V1), p(j, V1, Vf).
```

We run such a goal about `p` in the usual way (generating a CLP derivation tree) except that when we encounter the composite rule such as

```
p(i, V, Vf) :- loop(V, V1), p(j, V1, Vf).
```

we replace the embedded call to `loop` by the *result* of running `loop(V, V1)` in the current context. That is, we assume by induction, that a goal `?- Ψ, loop(V, V1)`, whose definition involves straightline predicates for level m, has previously been run. This results in constraints about the new variables `V1` and any loop invariant. Call these constraints $\Psi_1(V, V1)$. We therefore can replace the rule above

```
p(i, V, Vf) :- loop(V, V1), p(j, V1, Vf).
```

by

```
   p(i, V, Vf) :- Ψ1(V, V1), p(j, V1, Vf).
```

Repeating this process for all composite rules results in a CLP program which contains only basic rules for the one predicate `p`. The straightline algorithm now can be used to obtain the bound of the resource variable.

Consider now the `loop` predicates. Clearly each is executed at least as often as the program runs the corresponding loop in the worst case. We now show that the number is in fact proportional to the worst case actual run,

This follows essentially because of a *dynamic programming* formulation.

Suppose we are executing a goal $\mathcal{G}$ = `loop([...,Xs,...,T],[...,Tf])`. There are at most k reducts `loop_i([...,Xs,...,T1], [...,Tf])` where k is the number of intermittent invariants for this loop. Suppose each reduct is of the form

```
    T1 = T + αi, step(Xs, Xs1), loop_i([...,Xs1,...,T1], [...,Tf]).
```

where $1 \le i \le k$. Then clearly the value we seek is given by the recurrence:

$$time(\mathcal{G}) = \begin{cases} max \begin{Bmatrix} \alpha_1 + time(\mathcal{G}_1) \\ \cdots \\ \alpha_k + time(\mathcal{G}_k) \end{Bmatrix} & \text{if } \neg\texttt{exit(Xs)} \\ 0 & otherwise \end{cases}$$

which can be solved in a number of steps proportional to the number of steps that $\mathcal{G}$ need be unfolded until the exit condition is true.

Note that sometimes it is possible to evaluate the maximum time without having to unfold the loop. For example, in the recurrence

```
   loop([I, T],[_, T]) :- I >= n.
   loop([I, T],[_, Tf]) :- loop1([I+1, T + α1],[_, Tf]).
   loop1([I, T],[_, Tf]) :- loop([I+1, T + α2], [_, Tf]).
   loop([I, T],[_, Tf]) :- loop([I+1, T + β], [_, Tf]).
```

we can infer that the value of T in executing `loop([0, 0],[_, T])` is

$$\begin{array}{ll} 499*(\alpha_1+\alpha_2)+\alpha_1 & \text{if} \quad \alpha_1+\alpha_2 > 2*\beta \\ 999*\beta & \text{otherwise} \end{array}$$

Clearly such closed forms are not always easy to obtain, for example, when the total number of steps is not immediately known. As an extreme example, we recall the Collatz program whose closed form is not known.

Theorem 1. *Let the given program have a CLP representation P. Let r be the maximum number of rules in P at a given level. Let A be the algorithm employed for determining the resource bound for a straightline CLP program of r basic rules with performance complexity $f(r)$. Let a be the maximum number of intermittent invariants used in P. Then, the size of the computation tree explored by P is $O(a^2 * f(r))$.* ∎

We finally comment that for all practical purposes, the value of a, is to be considered constant because it is intended to be small.

```
expint: /* where n, x , and ITER are input (known statically) */
  if(x > 1) /* test */ { for (i=1;i<=ITER;i++){ } }
  else{
      for (i=1;i<=ITER;i++){
          if (i != n-1) {                   /* A */ }
          else{ for (j=1;j<=n-1;j++) { ...} /* B */ }}}
```

```
fft1:  /* where n is input (known statically)*/
  if (n < 2) { /* A */ }
  else{ xp2 = n;
        for(i = 0;i<log((double)n)/log(2.0); i++){/* loop i */
           xp = xp2; xp2 /= 2;
           for(j = 0; j < xp2; j++){              /* loop j */
              for(k = xp; k <= n; k += xp) {      /* loop k */}}}
        jj = 1;
        for(ii = 1; ii <= n-1; ii++){             /* loop ii */
          if(ii < jj)  { /* B  */  }
          kk = n/2;
          while(kk < jj){ jj -= kk; kk /= 2;}     /* loop kk */
          jj += kk;}}
```

Fig. 2. Relevant code for `expint` and `fft1`

6 Experimental Evaluation

In order to demonstrate the efficiency of our method we implemented our prototype bounds analyzer and performed a number of experiments. In these experiments we used real benchmarks from different sources. To show also the expressiveness power of our method, we will illustrate the most relevant pieces of some benchmarks and describe the main challenges for our analyzer. Our analyzer is implemented in CLP($\mathcal{R}$) system [11], and all measurements were performed on a Macbook Pro system with Intel Core Duo 1.83GHz CPU, 2 Gb RAM, and OS X 10.4.11.

For our first experiment we used two benchmarks: `expint` (Mälardalen benchmark suite [14]) and `fft1` (SNU-RT benchmark suite [21]). The results are shown in Table 1 and the important code is shown in Fig. 2. The column Parameter denotes the maximum number of iterations that depends on some key parameters. The Value column contains the exact length of the longest executable path inferred by our analyzer. The next two columns, States and Time, show the number of explored nodes during the execution of the path in the execution tree that gives rise to the longest path, and Time is the time in milliseconds. The last two columns denote the explored nodes and time also in milliseconds of our analysis. For the two benchmarks, all the input variable values are known at compile time by keeping the context using symbolic execution, hence the concrete states of the execution and the symbolic states inferred by our analysis were the same. As consequence, our analysis produced the exact length of the longest executable path, and more important, in time close to the simulation time. Note that in order to obtain this accurate result, the problems raised from the loops. In `expint`, the code section `B` displays a down-sampling since it is executed only once within the outer loop. On the other hand, the loops in `fft1` called `loop j` and `loop k` are non-rectangular loops since they depend on counter variables, `xp` and `xp2`, modified in the outer loop called `loop i`. Moreover, the code section `B` also displays a down-sampling because it is not always executed within the loop called `loop ii`. Finally we can observe that a

Benchmark	Parameter	Value	Analysis		Simulation	
			States	Time (ms)	States	Time (ms)
expint	ITER = 100	383	588	7.622	588	5.540
	ITER = 200	683	1088	16.637	1088	10.627
	ITER = 400	1283	2088	46.763	2088	32.978
	ITER = 800	2483	4088	161.678	4088	117.332
fft1	n = 4	77	130	1.339	130	0.994
	n = 8	352	461	14.466	461	11.562
	n = 12	4202	4371	2582.75	4371	2922.98

Table 1. Experimental Results for `expint` and `fft1`

```
calc_center: /* where image and N are input*/
  for(y=0;y<200;y ++){               /* loop y */
     for(x= y/2; x< 640 - y/2; x++){ /* loop x */
         if(image[x][y]){            /*    A   */
          black_pixel++; }   }}
  if (! black_pixel){                /*    B   */ }
```

Fig. 3. Relevant code for `calc_center`

closed-form for `loop kk` may be difficult to find since the lower and upper limits are modified at each iteration inside the loop.

For our second experiment shown in Table 2 we analyzed the program `calc_center` taken from Puschner and Koza [19] and shown partially in Fig. 3. This program has some interesting characteristics:

- The number of times that the code section A is executed depends on a variable N which is unknown at compile time.
- The two loops are non-rectangular loops. The loop called `loop x` depends on the counter variable of the outer loop (`loop y`).
- The code section A is down-sampling since it is known to be executed at most $N + \frac{N}{10}$ (from the problem specification and omitted in this code).
- Finally, A and the code section B are mutually exclusive.

Here we want to preserve the mutual exclusivity between sections A and B, as well as the correct number of iterations of both loops. The second problem is solved by maintaining the exact loop counter values (x and y). A naive solution to the first problem, would maintain the exact values of `black_pixel`. Unfortunately, this solution results in an exponential blowup of search space. We instead solve this problem using intermittent invariants. Here we provide the invariants `black_pixel=0` and `black_pixel>0`, which abstracts the values of `black_pixel`.

The experimental results are shown in Table 2. The parameters are constants used in the loop bounds. For example, in Fig. 3, the parameters are (200,640), corresponding to the constants used in the outer and inner `for` loops. Similar to previous experiments, States denote number of nodes and Time denotes running time. In Table 2 we compare the analysis and simulation results. As can be seen, the states visited in the analysis runs are constantly roughly four times those of the simulation runs, hence the exponential blowup is nonexistent. The slower running times of the analysis runs are due to our suboptimal implementation of memoing mechanism for already-visited states.

Parameter	Value	Analysis		Simulation	
		States	Time (ms)	States	Time (ms)
(5,16)	173	286	29.432	86	0.617
(10,32)	378	1106	238.544	301	2.993
(20,64)	1193	4366	3122.64	1136	17.610
(40,128)	4443	17366	57463.6	4426	116.606

Table 2. Experimental Results for `calc_center`

7 Concluding Remarks

We presented a general method for inferring and proving tight bounds on the resource usage of programs in which termination is guaranteed, and applied this result to the high-level analysis of Worst-Case Execution Time (WCET). The general method involves the use of intermittent invariants (i.e., each one is true for some but not all iterations of a loop) which allow the discrimination of different cases within a loop improving the accuracy of the overall method but still being practical. Since the automatic discovery of these invariants may be unfeasible, our method allows users to define intermittent invariants in a very flexible way and also checks invariants hold.

The other main contribution of this work is that we propose to unroll the loop to discover the bound. This is performed when the analysis of its body has been completed. This unrolling and optimization problem is amenable to dynamic programing, hence the cost is proportional to the actual running time of the loop and the number of intermittent invariants employed. In contrast, all prior methods need to discover a closed form expression for the loop, which is not just impossible in principle, but also impractical.

Finally, we have evaluated empirically our analysis with well-known benchmarks. Our method not only runs in good time, but produces accurate (and often *exact*) results.

References

1. On the 3x+1 problem. Available at http://www.ericr.nl/wondrous.
2. P. Altenbernd. On the false path problem in hard real-time programs. In *In Proceedings of the 8th Euromicro Workshop on Real-time Systems*, pages 102–107, 1996.
3. M. Rustagi C. A. Healy, M. Sjödin and D. Whalley. Bounding loop iterations for timing analysis. In *RTAS '98*, page 12. IEEE Computer Society, 1998.
4. A. Colin and G. Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *14th ECRTS*, page 50. IEEE Computer Society, 2002.
5. A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Syst.*, 18(2-3):249–274, 2000.
6. J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. *Real-Time Systems Symposium, IEEE International*, 0:163, 2000.
7. A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *3rd Euro-Par'97*, pages 1298–1307. Springer-Verlag, 1997.
8. A. Ermedahl, F. Stappert, and J. Engblom. Clustered calculation of pworst-case execution times. In *CASES '03*, pages 51–62. ACM, 2003.
9. C. A. Healy, R. D. Arnold, F. Mueller, M. G. Harmon, and D. B. Walley. Bounding pipeline and instruction cache performance. *IEEE Trans. Comput.*, 48(1):53–70, 1999.
10. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. LP*, 19/20:503–581, May/July 1994.
11. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.

12. J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *23rd AAAI*, pages 297–303. AAAI Press, 2008.
13. Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *SIGPLAN Not.*, 30(11):88–98, 1995.
14. Mälardalen WCET research group benchmarks. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, 2006.
15. G. Ottosson and M. Sjodin. Worst case execution time analysis for modern hardware architectures. In *In: Proc. of ACM SIGPLAN, Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 47–55, 1997.
16. C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Syst.*, 5(1):31–62, 1993.
17. C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, 24(5):48–57, 1991.
18. P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, 2000.
19. P. Puschner and C. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.
20. P.P. Puschner and A. V. Schedl. Computing maximum task execution times. *Real-Time Syst.*, 13(1):67–91, 1997.
21. SNU real-time benchmarks. URL `http://archi.snu.ac.kr/realtime/benchmark/`.
22. F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *J. Syst. Archit.*, 46(4):339–355, 2000.
23. F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *CASES '01*, pages 132–140. ACM, 2001.
24. E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *LCTES '01*, pages 88–93. ACM, 2001.
25. 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, Jan S., and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.