CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Dagstuhl Research Online Publication Server

COMPUTING TIME AS A PROGRAM VARIABLE: A WAY AROUND INFEASIBLE PATHS

Niklas Holsti[1]

Abstract

Conditional branches connect the values of program variables with the execution paths and thus with the execution times, including the worst-case execution time (WCET). Flow analysis aims to discover this connection and represent it as loop bounds and other path constraints. Usually, a specific analysis of the dependencies between branch conditions and assignments to variables creates some representation of the feasible paths, for example as IPET execution-count constraints, from which a WCET bound is calculated. This paper explores another approach that uses a more direct connection between variable values and execution time. The execution time is modeled as a program variable. An analysis of the dependencies between variables, including the execution-time variable, gives a WCET bound that excludes many infeasible paths. Examples show that the approach often works, in principle. It remains to be seen if it is scalable to real programs.

1. Introduction

Static WCET analysis is usually divided into three main parts [16]: *Flow analysis* models the possible execution paths (instruction sequences) as a control-flow graph (CFG). *Processor-behaviour analysis* bounds the execution time of each basic block in the CFG. *Bounds calculation* finds bounds on the execution time of entire execution paths. WCET tools often also make some analysis of the possible values of program variables. This *value analysis* supports the data-dependent parts of flow analysis and processor-behaviour analysis.

Define the *structural paths* as all paths through the CFG assuming that any conditional branch can be taken or not taken – as though the conditions could have any value, at any time. Most programs have loops and thus an infinite number of structural paths. Even after loop-bounds are applied to make a finite set of paths, often many logically infeasible paths remain, leading to an overestimated WCET bound. An *infeasible path* here means a connected sequence of CFG elements – nodes and edges – that contains an edge for which the condition must evaluate to *false* if the path is executed up to this edge and all earlier edge conditions on the path are *true*.

Flow analysis can detect some infeasible paths, *eg*. [2, 14]. Typically, the computations (expressions, assignments, branch conditions) are analysed and correlated to create some representation of the feasible and infeasible paths (*eg*. dead code or mutually exclusive basic blocks). This "flow fact" representation is then fed into the bounds calculation, for example as execution-count constraints for IPET ([6] and other references in [16]). The connection between the computation and the execution time is thus indirect: first from the computation to the flow facts, and then from the flow facts to the execution time. This paper explores another approach with a more direct connection. Section 2

1 Tidorum Ltd, Tiirasaarentie 32, FI 00200 Helsinki, Finland

discusses the dependencies between values computed in a program, and how these lead to infeasible paths. Section 3 presents the idea of the paper: to model the execution time as a program variable, subject to dependency-sensitive value analysis. Section 3 shows analysis results for some examples of infeasible paths, using a value analysis based on Presburger sets [12]. Section 4 describes this analysis for loop-less code. Section 5 extends the Presburger-set analysis to loops. Section 6 considers how the loop analysis handles the execution-time variable and infeasible paths involving loops. Section 7 closes the paper with a review of related work and a discussion.

2. Dependencies Between Variables and Values

Most variables in a program get their values from expressions that use the values of other variables, which leads to dependencies between the values of these variables. Dependencies also arise when different variables are independently computed and assigned within the same control structure. For example, in an **if-then-else** structure, the values assigned in the **then** branch occur together, and together with the *true* value of the condition, while those in the **else** branch occur together with the *false* value, but mixtures are infeasible.

Several static program analyses discover such dependencies; *eg*. [1]. Few WCET tools do it, but Lisper has suggested it [8]. Dependencies are the very reason for infeasible paths. Dependencies between loop induction variables and the looping condition lead to loop repetition bounds. The classic example of an infeasible path is two consecutive **if-then-else** structures with inter-dependent conditions.

The Bound-T WCET tool [5] models variable values as sets of integer tuples constrained by Presburger formulae – *Presburger sets* for short. Each element in a tuple models one variable; a tuple models one possible combination of variable values; and a set of tuples models all possible combinations of variable values. Dependencies between variables are included because the Presburger formulae constrain the whole tuple, not each element (variable) separately. For example, for two variables `x`, `y`, the set {[`x`, `y`] | `x` > 5 and `y` < 3`x`} models a state where `x` has any value greater than 5 and `y` has any value less than 3`x`. Operations on Presburger sets include set intersection, set union, set complement, relational join (mapping, application), convex hull, and test for subset [12]. Bound-T currently uses its Presburger model mainly for loop-bound analysis and does not search for other kinds of infeasible paths. This paper suggests how a dependency-sensitive value-analysis, such as the one in Bound-T, can be used to compute WCET bounds that exclude infeasible paths, without explicitly finding and representing the infeasible or feasible paths themselves. In fact, a separate bounds-calculation phase is not needed; value analysis produces WCET bounds.

3. Execution Time as a Computed Value

If we can analyse dependencies between variables, and we want to analyse the dependency between variables and execution time, perhaps we can get there by treating the execution time as a variable in the computation. To do so, augment the program (for the analysis only) with a new global variable, `T` say, that represents the execution time from the start of the program (or the subprogram, for a modular analysis). Augment each basic block *b* with the assignment `T` := `T` + *t*(*b*), where *t*(*b*) is the execution time of the basic block, as found by processor-behaviour analysis using the structural

paths. If the basic block can have a range of execution times, use interval arithmetic. Assume that `T` is initially zero. Use a value analysis to find the values of `T` at the end of the program; these are the execution times of the program. If the analysis models dependencies between variables, the final bounds on `T` should exclude some or all infeasible computations, depending on the precision of the analysis of the (indirect) dependencies between `T` and branch conditions.

3.1 Example: Condition after Condition

The classic example of infeasible paths is the correlated pair of **if-then-else** conditional statements:

```
if x < 1 then <compute for 100 cycles>;
         else <compute for 10 cycles>; end if;
<code that does not change x, for 30 cycles>;
if x > 3 then <compute for 200 cycles>;
         else <compute for 20 cycles>; end if;
```

(For clarity, the time for evaluating the conditions is included in the times for the **then** and **else** branches.) Each conditional statement in this example has a fast branch and a slow branch. The structural paths include the path that takes both the slow branches, totalling 100 + 30 + 200 = 330 cycles. However, this path is logically infeasible because the conditions for the two slow branches, $x < 1$ and $x > 3$, cannot be true at the same time. The longest feasible path occurs when $x > 3$ and gives 10 + 30 + 200 = 240 cycles. Now augment the code with the execution-time variable `T` :

```
T := 0;
if x < 1 then <compute for 100 cycles>;  T := T + 100;
         else <compute for 10 cycles>;   T := T + 10; end if;
<code that does not change x, for 30 cycles>;  T := T + 30;
if x > 3 then <compute for 200 cycles>;  T := T + 200;
         else <compute for 20 cycles>;   T := T + 20; end if;
```

A value analysis of the augmented program with the Presburger method (to be described in Section 4) gives a model of the final values of the variable tuple [x, T] as the three-part set:

$$\{[x, T] \mid x < 1 \text{ and } T = 150\} \cup \{[x, T] \mid 1 \le x \le 3 \text{ and } T = 60\} \cup \{[x, T] \mid x > 3 \text{ and } T = 240\}$$

The analysis thus excludes the infeasible value $T = 330$ and gives the precise bounds $T = 60 \,..\, 240$.

3.2 Example: Saturating a Value

This example code makes sure that the variable `x` does not exceed the interval `min` .. `max`:

```
if x < min then x := min; end if;
if x > max then x := max; end if;
```

This code has an infeasible path if `min ≤ max`, because the assignment `x := min` makes the condition in the second **if** false. (The infeasible path could be avoided with an **else if**, but some programmers seem to prefer the above form.) Augment the program with an execution-time variable `T` as follows, adding **else** branches just to model the condition-evaluation time:

```
T := 0;
if x < min then x := min; T := T + 3; else T := T + 1; end if;
if x > max then x := max; T := T + 3; else T := T + 1; end if;
```

Assume for simplicity that `min` = 1 and `max` = 10. The model of the final values of [`x`, `T`] is then:

$$\{[x, 2] \mid 1 \le x \le 10\} \cup \{[1, 4]\} \cup \{[10, 4]\}$$

Thus, the infeasible value `T` = 6 is avoided, and the precise range `T` = 2 .. 4 is computed. The correct range for `T` is computed even if the actual values of `min` and `max` are unknown (variable), as long as the analyser knows that `min` ≤ `max`. The path-exclusion analysis of Stein and Martin [14] cannot handle this example because the assignment `x` := `min` does not dominate the second conditional. However, as Stein and Martin say, their analysis could be extended to such "non-linear slices" at the cost of more complex Presburger problems and increased analysis time.

4. Presburger Analysis Basics

This section describes the Presburger-set value-analysis that gives the above results, taking the saturation code in section 3.2 as an example. The analysis is similar to the one in Bound-T [5].

In the absence of loops the analysis is quite simple. Assume that there are n variables. At the start of the program the model is the universal set of all possible variable-value tuples Z^n, where Z is the set of all integers. Our example has the variables `x` and `T` so $n = 2$.

Each statement (in Bound-T: each machine instruction) is modeled as a transfer relation between the n input values and the n output values, a Presburger subset of $Z^{2n} = Z^n \times Z^n$. For example, the statement `T` := 0 is modeled by the transfer relation {[`x`, `T`, `x'`, `T'`] | `x'` = `x` and `T'` = 0}. Here the symbols `x` and `T` represent the variable values before the statement, while the primed symbols `x'` and `T'` represent the values after the statement. This relation can be written more compactly as {[`x`, `T`] → [`x`, `0`]} where the arrow separates the "before" and "after" values and the constraints are implied by the expressions. The universal 2-tuple set Z^2 is then written {[`x`, `T`]}.

Starting from the universal set {[`x`, `T`]} and applying the transfer relation {[`x`, `T`] → [`x`, `0`]} produces the set of new variable values {[`x`, `0`]}, which models the variable values after the statement `T` := 0. In this way, the variable-value model is propagated over the statements and nodes in the CFG. To propagate the model over a CFG edge, the model set is intersected with the set defined by the edge condition. When several edges converge to the same node, the union of the sets from each incoming edge is computed. The union set is often non-convex, which increases both the power and the complexity of the analysis, and differs from the model in [1]. Condition flags are another source of non-convex sets, because their values are constrained by disjunctive formulas.

Figure 1 illustrates this analysis for the example in section 3.2. The figure shows the CFG with Presburger sets placed before and after nodes to show the value-model at these points. Note the edge from the second **if** towards the right, when `x` > 10; this edge condition eliminates the tuple [1, 3] from the value set, which reflects the mutual exclusion of the two CFG nodes on the right. If the analysis gives an empty Presburger set at some point – which does not happen in this example – that part of the CFG is infeasible (dead).

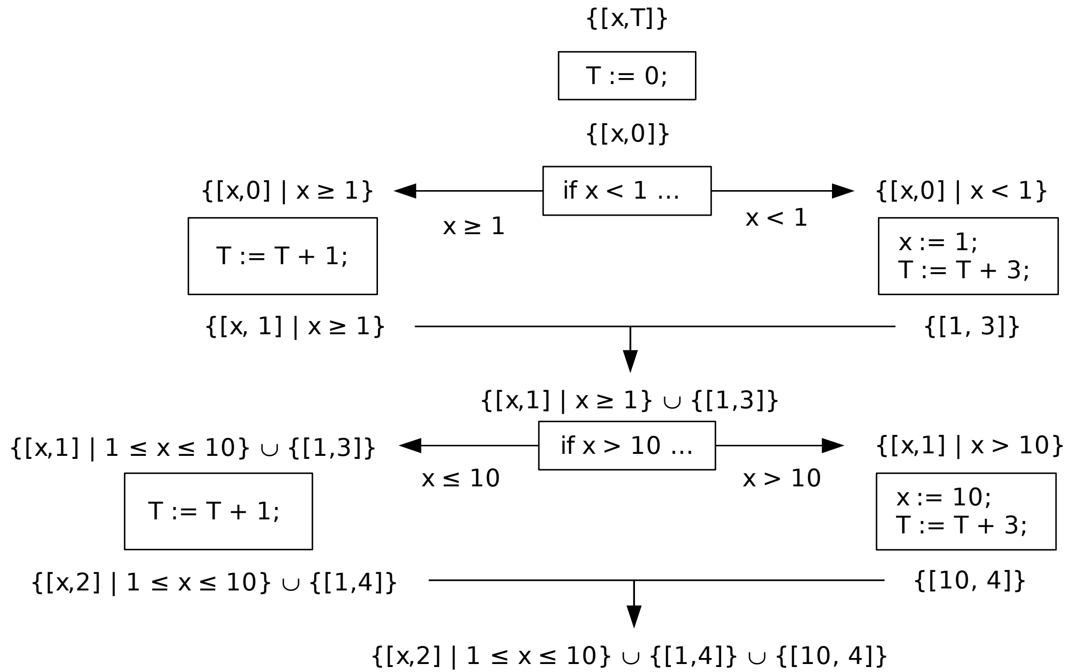

Figure 1: Presburger analysis of the saturation example

Computation with Presburger sets has exponential complexity in general. For practical purposes it is important to minimize the number of constraints and union operations. If the `T` variable is updated separately in each branch of a conditional, as above, `T` depends on every branch condition. For a conditional where the branches have similar execution times, one could instead update `T` once for the whole conditional, with `T := T +` *max* (**then** branch, **else** branch), as in the *timing schema* methods [13]. The complexity can thus be reduced by a selective fall-back to a timing schema.

5. Presburger Analysis of Loops

This section extends the Presburger-set analysis to loops, showing how it can find bounds on the number of iterations and bounds on the variables after the loop. As a running example, consider this code that reverses the order of elements `n` .. `n` + 9 of the vector `vec`, where `n` is a variable:

```
i := n; j := n + 9;
while i < j loop
   z := vec[i]; vec[i] := vec[j]; vec[j] := z;
   i := i + 1; j := j – 1;
end loop;
```

The analysis will ignore the values in `vec` because they have no effect on the loop or its execution time, and pointer analysis is out of scope for this paper. The modeled variables are `i`, `j`, `n`, and `z`. Since the `vec` values are ignored, the assignment to `z` is modeled as storing an unknown value in `z`.

Assume, initially, that there is only one loop, with a single *loop head node* that is the single point of entry to the loop. The analysis is based on the *repetition relation* of the loop. One pass through the loop body from the loop head up to and including a back edge is called a *repetition* of the loop. The

repetition relation is the Presburger relation that shows how one repetition changes variable values. The repetition relation of the example loop above is {[i, j, n, z] → [i+1, j-1, n, z'] | i < j}, where z', the new value of z, is unconstrained. Similarly, a pass through the loop body from the loop head to an edge that leaves the loop is called an *exit* from the loop. The full execution of a loop is a sequence of zero or more repetitions, followed by one exit.

5.1 Classifying Variables as Invariant, Induction, or Fuzzy

For a program with loops, static analysis needs some form of induction or least-fixpoint iteration. The Presburger-set domain has infinite ascending chains, so least-fixpoint iteration cannot be used directly. Instead, we analyse the repetition relation of the loop to divide the variables into three classes: an *invariant* variable is not changed at all; an *induction* variable is changed by a bounded *increment* (possibly negative); and the rest are *fuzzy* variables that change in other ways.

Let R be the repetition relation of a loop and x a variable. Introduce a new variable dx on the domain side to represent the possible increment in x, making R a relation in $Z^{n+1} \times Z^n$. This role of dx is expressed by the relation Rd = R ∩ {[..., x, ..., dx] → [..., x + dx, ...]}. Compute bounds on dx by taking the domain of Rd, then projecting away the other variables to leave only the set of dx values, and finally computing the convex hull of this one-dimensional set. The convex hull is the interval that bounds dx. If dx is bounded to zero, x is invariant; otherwise, if dx is bounded to a finite interval, x is an induction variable; otherwise x is a fuzzy variable. Note that dx can depend on the other variables, but the projection and convex-hull operations discard those dependencies in order to arrive at *constant* lower and upper bounds on dx. This is a necessary approximation at this step of the analysis because the next step uses the bounds on dx to multiply a variable, and the Presburger model allows multiplication only when at most one of the factors is a variable.

It is practical to introduce all increment variables at once. In the vector-reversal example, this introduces the variables di, dj, dn, and dz and gives the extended loop repetition relation

Rd = {[i, j, n, z, di, dj, dn, dz] → [i+1, j-1, n, z'] | i < j}
∩ {[i, j, n, z, di, dj, dn, dz] → [i+di, j+dj, n+dn, z+dz]}

Computing bounds on di, dj, dn, dz from this Rd shows that di = 1, dj = -1, dn = 0, and dz is not bounded. Thus i and j are induction variables, n is invariant, and z is a fuzzy variable.

5.2 Bounding the Number of Loop Repetitions

Introduce a new variable c to model the iteration number 0, 1, 2, ... The value-model at the start of the loop-head node is a Presburger set where each invariant variable has its initial (and invariant) value, each induction variable equals its initial value plus c times its increment, and all fuzzy variables are unconstrained[2]. Propagate the model from the loop head to all parts of the loop as in section 4. Next, check if the Presburger sets on the back edges or exit edges of the loop (which include the repetition or termination conditions) imply bounds on c; if so, these are the loop repetition bounds.

2 A better but more complex model includes variable dependencies from the preceding iteration. Bound-T does so.

For the vector-reversal example, the Presburger set on the loop back edge is

$$\{[i, j, n, z, c] \mid i-1 < j+1 \text{ and } i = n+1+c \text{ and } j = n+8-c\}$$

Here `i` and `j` represent the variable values at the end of the loop body, after they have been incremented by `di` and `dj`. Projecting this set to the variable `c` and computing the convex hull gives the constraint $c \le 4$, which shows that the loop is repeated at most 5 times (for $c = 0 .. 4$). Of course, if the number of repetitions of a loop is bounded by a user annotation, it is unnecessary to compute bounds on `c`, as they are given by the annotation.

5.3 Modeling Variables After the Loop

The final step of the Presburger analysis of a loop is to model the variable values after the loop as the result of some number of loop repetitions followed by a loop exit. For an induction variable, the repetitions are modeled by multiplying the number of repetitions with the increment of the variable. Both factors have constant bounds, but simply multiplying these constants would hide possible dependencies between the increments, the number of loop repetitions, and other variables. At this point there is a choice: one, but not both, of the factors can be treated as a Presburger variable, making the analysis sensitive to dependencies between this variable and other variables. I have not found a way to compute the number of repetitions as a Presburger variable, only as a constant extracted from the Presburger model (bounds on the iteration number `c`), and therefore I choose the increment factor as the Presburger variable, leaving the number of loop repetitions as a constant factor. In the subsequent analysis of the post-loop code, the auxiliary variables for the increments can be projected away; their relationships, if any, are still encoded in the structure of the projected set.

In the vector-reversal example, there are no dependencies between the increments and other variables. Similar analysis (omitted here) shows that the minimum number of repetitions is also 5, so the values after the loop are modeled by the set $\{[i, j, n, z] \mid i = n+5 \text{ and } j = n+4\}$.

For a loop that has more than one entry point, in other words more than one head node, a repetition is defined as a pass through the loop from some head node to some back edge, and an exit is defined as a pass from some head node to some edge that leaves the loop. The repetition relation is defined as the union of the transfer relations from all possible repetition paths, and the set of initial values as the union of the initial-value sets at all head nodes. The analysis itself remains the same.

If there are nested loops the Presburger-set analysis has two phases. The first phase analyses the loops bottom-up, from the innermost to the outermost, classifies the variables as invariant, induction, or fuzzy for each loop, and makes an approximate model of the total effect of each loop as a relation that uses this classification but an unknown number of loop repetitions. The repetition relation of an outer loop uses these approximations for all inner loops. The second phase analyses the loops top-down, from outermost to innermost, and in flow order for loops at the same level. This second phase computes loop-repetition bounds and better post-loop value models that include bounds on the induction-variable increments and bounds on the number of repetitions. In some cases the models can be further improved by iterating these two phases a few times. Note also that user annotations that place bounds on variable values, at one program point or everywhere, are quite

easy to include in the Presburger sets, apart from the practical problem of mapping source-level variable identifiers to machine-level storage locations.

6. Execution Time of Loops

This section considers how the Presburger-set analysis of loops applies to the execution-time variable `T`, and in particular whether the bounds on `T` include or exclude infeasible paths in loops. The execution-time variable `T` is evidently an induction variable, and its increment `dT` is the execution time of the loop body. From section 5 the model of `T` after the loop is the initial value T_0 of `T` (the cumulated execution time up to the start of the loop) plus `dT` times the number of loop repetitions, plus the execution time of loop exit. This is roughly the same formula as in the timing schema methods [13] but here both T_0 and `dT` can depend on other variables. The exit from a loop is analysed using the methods in section 4 and is already "outside" the loop from the viewpoint of loop analysis. Thus, only loop repetitions are discussed here.

Loops can involve many kinds of dependencies and infeasible paths. For some kinds, the proposed analysis excludes infeasible paths; for others it does not, as discussed below.

6.1 Iteration-Independent Dependencies

A dependency or an infeasible path is *iteration-independent* if it applies on every repetition of the loop. In the simplest case, the loop contains a path that is infeasible on every repetition, whatever happens outside the loop or in other repetitions or in an exit from the loop. An example is a loop that contains the "saturation" code from section 3.2. The Presburger analysis omits the infeasible path from `dT`, and so also from the final value of `T`.

Another example is a conditional outside the loop that conflicts with a conditional inside the loop, on every repetition of the loop. For example, put a loop around the second **if-then-else** statement in the example in section 3.1, keeping the variable `x` as a loop invariant, thus:

```
t := 0;
if x < 1  then t := t + 100;
          else t := t +  10; end if;
for i in 1 .. 7 loop
   if x > 3  then t := t + 200;
             else t := t +  20; end if;
end loop;
```

In the repetition relation Rd the `T`-increment `dT` now depends on `x`, as can be seen in the domain of Rd projected to the variables [`x`, `dt`], which is the set $\{[x, 20] \mid x \le 3\} \cup \{[x, 200] \mid x > 3\}$. If execution takes the slow branch of the first conditional, before the loop, it must then take the fast branch of the second conditional, inside the loop, on every loop iteration. This dependency is reflected in the model of the values after the loop. Projected to the variables [`x`, `T`] this set is

$$\{[x, 240] \mid x < 1\} \cup \{[x, 150] \mid 1 \le x \le 3\} \cup \{[x, 1410] \mid 4 < x\}$$

The infeasible path where both slow branches are taken would give a final `T` of 100 + 7 × 200 = 1500 cycles, which this analysis evidently excludes. The worst case takes the fast branch before the loop and the slow branch in every repetition of the loop, giving 10 + 7 × 200 = 1410 cycles.

6.2 Iteration-Specific Dependencies

Some dependencies and infeasible paths apply only on some repetitions. For example, put a loop around the second **if-then-else** statement in the example in section 3.1, but also add a statement in the loop that changes `x` on repetition 4. Now the combination of the two slow branches is infeasible for repetitions 0 through 3, but may be feasible for later repetitions. The Presburger analysis cannot exclude these infeasible paths because the bounds on `dT` are computed over all repetitions together.

When a loop contains important iteration-dependent "unbalanced" conditionals, it is possible to introduce new variables to model how often each branch is executed, and compute the final `T` as the sum of these execution frequencies times the execution times (`dT`'s) of the branches. This is very similar to the IPET formulation, but now the `dT`'s of the branches can depend on other variables, so iteration-independent infeasible paths can be excluded. The execution frequencies could perhaps be computed analytically from the Presburger model [11] or by other methods [4, 8].

6.3 Dependent Loop Bounds

Important infeasible paths may arise when loop bounds correlate with conditionals, as here:

```
if <cond> then <compute for 100 cycles>; n :=  5;
          else <compute for  10 cycles>; n := 20; end if;
for i in 1 .. n loop <compute for 10 cycles>; end loop;
```

A normal WCET analysis includes the infeasible path where the slow **then** branch is followed by 20 iterations of the loop, for a total of 300 cycles, although this branch implies only 5 iterations and 150 cycles. The actual WCET is 210 cycles, for the fast **else** branch followed by 20 iterations. The analysis of this paper does no better and reports a WCET bound of 300 cycles, because the bound on the number of loop repetitions is modeled as a constant (20), not as a Presburger variable (`n`) that can depend on other variables. As explained in section 5.3 the loop bound and the increment `dT` cannot *both* be Presburger variables, because the Presburger model does not allow multiplication of two variables. Thus, one cannot at the same time model the dependencies of `dT` and of the loop bound.

7. Discussion

The two main points in this paper are to model execution time as a program variable, and to use a dependency-sensitive value-analysis to exclude infeasible values. Neither point is novel in itself, but the combination is new, as far as I know. Haase [3] defined execution time as a program variable, for proving real-time properties using weakest-precondition calculus. Haase's work influenced Shaw's definition of timing schemata [13], but the *max* (**then**, **else**) schema for conditionals disconnects execution time from other variables. The annotation language described by Mok *et al.* [9] is a kind of program to compute the execution time, but it is evaluated apart from the target program.

Model-checking approaches to timing analysis [10] must treat execution time as a variable, and may use semi-symbolic state representations. The analysis of Cousot and Halbwachs [1] identifies affine dependencies between variables, but uses convex hulls, which I believe means that its ability to exclude infeasible paths is weak. Lisper included dependency-sensitive value analysis in his ambitious proposals [8], but without execution time as a variable. Stein and Martin use Presburger models in their analysis of exclusive paths [14]. However, they create constraints for bounds calculation by IPET, and do not use a time variable.

The method explored in this paper needs a name. Let us call it the *time-variable* method. Different value analyses can be used with this method; the Presburger analysis is only one possibility. The examples show that the time-variable method with Presburger analysis excludes some kinds of infeasible paths, but not all kinds. The method is in principle applicable to more kinds of of infeasible paths than some other methods, such as [14]. The time-variable method does not need a representation of feasible and infeasible paths (a flow-fact language) nor a separate bounds-calculation phase. A drawback is that it does not, by itself, exclude infeasible paths from the processor-behaviour analysis. This could lead to over-estimated WCETs of CFG elements. However, while it is easy to exclude single infeasible (dead) nodes and edges from the processor-behaviour analysis, it seems much harder to exclude longer infeasible paths, because the analysis would have to identify which paths generate which processor states. Hence this drawback may be insignificant.

In processors with caches and other accelerators the execution time $t(b)$ of a basic block b is not constant but depends on the history of execution. Using a history-independent worst-case $t(b)$ can be a gross over-estimation. Current WCET tools have two approaches to this problem. The first approach (virtually) expands the CFG so that the same basic block is represented by different CFG nodes in different contexts, and thus by different context-specific values of $t(b)$ [15]. The most common expansion peels the first iteration from each loop so as to separate the I-cache misses that may occur on the first iteration, from the I-cache hits that are assured on later iterations. The time-variable method allows such CFG expansions and their benefits are the same as in the IPET method.

The second approach to handle history-sensitive accelerators is to expand the IPET problem by new ILP variables that model the states of the accelerator mechanisms (*eg*. the contents of the I-cache) and the additional execution time required by accelerator state transitions (*eg*. an I-cache miss) [7]. As the time-variable method does not use IPET, it cannot use this approach as such. In principle, new Presburger variables could model accelerator states, but the complexity would likely be impractical.

Experience with the Bound-T tool [5] shows that the Presburger analysis is practical in many cases for analysis of loop bounds, but also impractical in some cases (large subprograms), at least in its current implementation. I think that the complexity comes mainly from the disjunctions (set unions). In [14] Stein and and Martin restrict their analysis to “linear” slices and thus avoid disjunctions. Still, I believe that disjunctions are necessary for analysing several kinds of infeasible paths. Better slicing to simplify the Presburger model, selective use of a timing schema for well-balanced conditionals, and perhaps taking the convex hull of the Presburger sets at selected program points may reduce the analysis time. Bound-T currently calculates WCET bounds with IPET, not with the time-variable method. Implementation and experimental evaluation of the time-variable method is future work.

8. References

[1] COUSOT, P., HALBWACHS, N., Automatic discovery of linear restraints among variables of a program, in: Proc. of the 5th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages, January 1978.

[2] GUSTAFSSON, J., ERMEDAHL, A., SANDBERG, C., and LISPER, B., Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution, in: Proc. of the 27th IEEE International Real-Time Systems Symposium (RTSS'06), December 2006.

[3] HAASE, V.H., Real-time behaviour of programs, in: IEEE Transactions on Software Engineering, Vol SE-7, No. 5, September 1981.

[4] HEALY, C., SJÖDIN, M., RUSTAGI, V., WHALLEY, D., and ENGELEN, R. V., Supporting Timing Analysis by Automatic Bounding of Loop Iterations, in: Real-Time Systems, vol. 18, no. 2-3, May 2000.

[5] HOLSTI, N., Bound-T Reference Manual. Tidorum Ltd, http://www.bound-t.com/, February 2008.

[6] LI, Y.-T. S., MALIK, S., Performance analysis of embedded software using implicit path enumeration, in: Proc. of the 32:nd Design Automation Conference. 456–461, 1995.

[7] LI, Y.-T. S., MALIK, S. WOLFE, A., Cache modeling for real-time software: beyond direct mapped instruction caches, in: Proc. of the 17th IEEE Real-Time Systems Symposium, December 1996.

[8] LISPER, B., Fully automatic, parametric worst-case execution-time analysis, in: Proc. of the 3rd International Workshop on Worst-Case Execution Time Analysis, Porto, Portugal, July 2003.

[9] MOK, A.K., AMERASINGHE, P., CHEN, M., and TANTISIRIVAT, K., Evaluating tight execution time bounds of programs by annotations, in: Proc. 6th IEEE Workshop on Real-Time Operating Systems and Software, Pittsburgh, PA, USA, May 1989.

[10] NAYDICH, D., GUASPARI, D., Timing analysis by model checking, in: Lfm2000 – Fifth NASA Langley Formal Methods Workshop, Technical Report NASA-2000-cp210100.

[11] PUGH, W., Counting solutions to Presburger formulas: how and why, in: Proc. of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, Orlando, Florida, United States, June 1994.

[12] PUGH, W., et al., The Omega project: frameworks and algorithms for the analysis and transformation of scientific programs, University of Maryland, http://www.cs.umd.edu/projects/omega.

[13] SHAW, A.C., Reasoning about time in higher-level language software, in: IEEE Transactions on Software Engineering, Vol 15, No 7, July 1989.

[14] STEIN, I., MARTIN, F., Analysis of path exclusion at the machine code level, in: Proc. of the 7th International Workshop on Worst-Case Execution Time Analysis, Pisa, Italy, July 2007.

[15] THEILING, H., FERDINAND, C., Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis, in: Proc. of the 19th IEEE Real-Time Systems Symposium, December 1998.

[16] WILHELM, R., et al., The worst-case execution time problem — overview of methods and survey of tools, in: ACM Transactions on Embedded Computing Systems, Volume 7, Issue 3, April 2008.