Advance Access publication on September 26, 2007 doi:10.1093/comjnl/bxm058

Automated Worst-Case Execution Time Analysis Based on Program Modes

MENG-LUO JI[1], JI WANG[2,*], SHUHAO LI[2] AND ZHI-CHANG QI[1]

[1]*Department of Computer Science, National University of Defense Technology, Changsha, China*
[2]*National Laboratory for Parallel and Distributed Processing, Changsha, China*
**Corresponding author: wj@nudt.edu.cn*

A program mode is a regular trajectory of the execution of a program that is determined by the values of its input variables. By exploiting program modes, we may make worst-case execution time (WCET) analysis more precise. This paper presents a novel method to automatically find program modes and calculate the WCET estimates of programs. First, the modes of a program will be identified automatically by mode-relevant program slicing, and the precondition will be calculated for each mode using a path-wise test data generation method. Then, for each feasible mode, we show how to calculate its WCET estimate for modern reduced instruction set computer (RISC) processors with caches and pipelines and for traditional complex instruction set computer (CISC) processors. We also present a method to obtain the symbolic expression for each mode for CISC processors. The experimental results show the effectiveness of the method.

Keywords: real-time systems; WCET analysis; program mode; program slicing; iterative relaxation method

Received 4 October 2006; revised 31 May 2007

1. INTRODUCTION

Calculation of the worst-case execution time (WCET) of tasks is of prime importance in the timing analysis of real-time systems. The purpose of WCET analysis is to estimate *a priori* WCET of a piece of code on a given processor. A main issue in WCET analysis is to avoid pessimism in the evaluation processes [1]. Precise estimates of WCET enables better budgeting and scheduling of system resources.

It is noticed that routines, tasks and main programs (all referred to as programs in this paper) usually have modes. A mode is a regular trajectory or trace of the execution of a program that is determined by the values of its input variables. With these inputs, the program executes along that trace. Consider the example code segment [2] in Fig. 1.

The function `pow` computes the power of a floating point numeral $\texttt{base}^{\texttt{exponent}}$. If the parameter `exponent` in function`pow` is not less than zero, then the result is $\texttt{base}^{\texttt{exponent}}$, otherwise the result is $1/\texttt{base}^{-\texttt{exponent}}$. This program executes along different trajectories depending on whether the value of input parameter `exponent` is negative. Therefore, the WCET estimate depends on the range of `exponent`. Bernat and Burns [2] show the result of the WCET estimate of the example:

$$\mathrm{WCET}_{\texttt{pow}}(\texttt{exponent})=\begin{cases}1752-434*\texttt{exponent}, & \texttt{exponent}<0\\ 1474+434*\texttt{exponent}, & \texttt{exponent}\geq 0\end{cases}$$

In the example of Fig. 1, the function `pow` has two modes: one when `exponent` is negative and the other when `exponent` is nonnegative. Furthermore, when a program has more than one mode, it may have different WCET estimates (in the forms of concrete values or parametric formulas) under different modes. Because the preconditions of modes can quickly be evaluated and the decisions can be made when scheduling a task at run time, or calling a routine under some kinds of contexts, the calculation of the WCET for each mode of a program can make the WCET estimate of the whole program more precise.

For example, let us consider the following code segment:

```
1. result=0;
2. for (i=-2;i<8;i++)
3.   result=result+Pow(3,i);
```

```
float Pow (float base, int exponent){
 1.  float result = 1.0;
 2.  int times;
 3.  bool exchange;
 4.  exchange = true;
 5.  if (exponent < 0)
 6.    times = -exponent;
 7.  else {
 8.    exchange = false;
 9.    times = exponent;
10. }
11. for (i = 0;i < times;i++)
12.   result = result*base;
13. if (exchange)
14.   result = 1 / result;
15. return result;
16.}
```

FIGURE 1. Example code segment #1.

The total WCET estimates of calling the function `pow` within the body of `for` statement should be

$$\sum_{i=-2}^{-1}(1752 - 434^{*}i) + \sum_{i=0}^{8}(1474 + 434^{*}i).$$

However, if we do not take the modes into account, the corresponding estimate would at least be

$$\sum_{i=-2}^{8}(1752 + 434^{*}|i|).$$

The difference shows that program modes can be used to calculate a tighter WCET estimate of a whole program.

In this paper, we present a novel method to automatically obtain the modes of a program and calculate the WCET estimate for each given mode. Basically, it consists of two phases. In phase one, we construct a new program by mode-relevant slicing, a variant of program slicing [3, 4]. We list all the paths of the sliced program. Each path in the sliced program either corresponds to a mode or is an infeasible path. For each path, by applying the iterative relaxation method [5, 6], in most cases, we may compute the precondition for the path, or assert that the path is infeasible. The precondition of a path is a set of constraints on the input variables that guarantee the program to execute along the path. In phase two, we calculate the WCET estimate for each given mode for reduced instruction set computer (RISC) and complex instruction set computer (CISC) processors, respectively.

The remainder of this paper is organized as follows. The preliminary concepts about program analysis are introduced in Section 2. Section 3 describes the mode-relevant program slicing technique. Section 4 gives an algorithm to generate the precondition for each mode. Section 5 presents a WCET analysis method for a program mode for RISC as well as CISC processors. Section 6 presents a WCET analysis method for calculating symbolic expressions for each mode for CISC processors. Section 7 describes our prototype tool. Section 8 reports on some experiments on RISC processors with the tool. In Section 9, we discuss the related work and conclude the paper.

2. PRELIMINARIES

In this paper, a program M is viewed as a directed control flow graph CFG $= (N, E, entry, exit)$, where N is a set of nodes, E a set of edges, *entry* a unique entry node and *exit* a unique exit node of M. A node $n \in N$ represents a single statement or a conditional expression. A possible transfer of control from node n_i to node n_j is mapped to an edge $(n_i, n_j) \in E$. A path $P = <n_1, n_2, \ldots, n_{k+1}>$ in CFG is a sequence of nodes such that $(n_i, n_{i+1}) \in E$, for $i = 1, \ldots, k$. The length of the path P, denoted by $|P|$, is the number of nodes on the path.

Let V be the set of all variables that are referenced in M. A variable in V is an *input variable* of M if it either appears in an input statement of M or is an input parameter of M. The domain D_k of input variable i_k is the set of all possible values, which may be assigned to i_k. An input vector $I = <i_1, i_2, \ldots, i_m> \in (D_1 \times D_2 \times \cdots \times D_m)$ is called a *program input* or *input*, where m is the number of input variables.

A multi-way decision statement appears in CFG as a *branch node*. The conditional expression of a branch node is called a *branch predicate* (or simply *predicate*). Here, we assume that the branch predicates are simple relational expressions (inequalities and equalities) of the form E_1 *op* E_2, where E_1 and E_2 are arithmetic expressions, and *op* is one of $\{<, \leq, >, \geq, =, \neq\}$.

For a branch node m, let $(m, n) \in E$ be a *branch*, $Cond(m, n)$ be the condition under which the control is transferred from m to n. That is, node m traverses branch (m, n) when $Cond(m, n) =$ TRUE.

Each branch predicate E_1 *op* E_2 can be transformed into an equivalent branch predicate of the form F *op* 0, where F is an arithmetic expression $E_1 - E_2$. Along a given path, F represents a real-valued function called a *predicate function.* F may be a direct or indirect function of input variables.

Each node n (i.e. each statement in the program or node in CFG, we do not differentiate them in this paper) is associated with two sets: $Ref(n)$, the set of variables whose values are referenced at n, and $Def(n)$, the set of variables whose values are defined at n.

A node n in CFG is *post-dominated* by a node m if all the paths from n to *exit* pass through m [7]. A node n is *control dependent* on a node m if (i) there exists a path P from m to n with any node u, $u \neq m$ and $u \neq n$, in P, u is post-dominated by n; (ii) m is not post-dominated by n [7]. For the programs with structured control flow (there are no goto statements in program M), the statements in the branches of a predicate b are control dependent on predicate b. The nodes in the body of a loop structure are defined as loop controlled nodes.

The range of influence of a branch statement b, $Infl(b)$, is defined as the set of statements that are control dependent on b.

3. MODE-RELEVANT PROGRAM SLICING

A mode of a program keeps the program to execute in the regular pattern or trajectory that is determined by the values of input variables. To identify the modes, it is needed to examine the predicates that are totally dependent on input variables. These predicates are called *input-dependent predicates*. A predicate n is an input-dependent predicate if $\forall x \in Ref(n)$, x is an input variable or x can directly be calculated using input variables. In this section, we will extract the input-dependent predicate-relevant statements from the program to form a new program, which is called a *mode-relevant slice*. The basic idea is to slice the program in a forward way after determining the input-dependent variables (i.e. the variables which are totally dependent on input variables) of the program using a data-flow framework [8].

3.1. Determining input-dependent variables

Let σ be a function that maps the variables in V to the values of a specific value set SV. Here, σ is called an abstract state. For a statement n, the state before the execution of n and the state after the execution of n are denoted as $\sigma^n_\circ$ and $\sigma^n_\bullet$, respectively. The specific value set SV is defined as $SV = \{InDep, Undef, LCtrl\}$, where the state of a variable is $InDep$ if it is totally dependent on input variables and the state of a variable is $LCtrl$ if the variable is controlled by a loop, i.e. its value is dependent on a loop. Initially, the states of all variables are $Undef$ by default. We define a total ordering $\sqsubseteq$ over the specific value set as $LCtrl \sqsubseteq InDep \sqsubseteq Undef$.

Our iteration algorithm for determining input-dependent variables consists of two steps in addition to initialization, as shown in Fig. 2.

Initialization. For node *entry*, the input variables are assigned $InDep$, other variables are assigned $Undef$. For any statement n other than *entry*, the variables are assigned $Undef$.

In the first step, a dataflow framework is used to analyze the program.

(i) For each node n, if variable $x \in Def(n)$ and $\sigma^n_\circ(x) = LCtrl$ then $\sigma^n_\bullet(x) = LCtrl$, else

$$\sigma^n_\bullet(x) = \begin{cases} InDep, & \text{if } x \in Def(n) \text{ and } \forall y \in Ref(n), \\ & \sigma^n_\circ(y) = InDep, \\ \sigma^n_\circ(x), & \text{otherwise.} \end{cases}$$

The state of a variable x after node n is defined $InDep$ only when x is defined at n and the states of all the variables referenced at n is $InDep$.

INPUT:
N - the set of nodes;
E - the set of edges;
The identification of input variables and loop predicates.
OUTPUT:
the state of each variable at each point of the program.
// Initialization:
IF x is an input variable **THEN**
$\sigma^{entry}_\circ(x) = InDep$
ELSE
$\sigma^{entry}_\circ(x) = Undef$
ENDIF
FOREACH $n \neq entry$ **DO**
$\sigma^n_\circ(x) = Undef$
ENDFOR
// Step 1:
WHILE the state of any variable is changed **DO**
FOREACH $n \in N$ **DO**
IF $(x \in Def(n))$ && $(\sigma^n_\circ(x) = LCtrl)$ **THEN**
$\sigma^n_\bullet(x) = LCtrl$
ELSEIF $x \in Def(n)$ **THEN**
BOOL AllTrue = TRUE
FOREACH $y \in Ref(n)$ **DO**
IF $\sigma^n_\circ(y) \neq InDep$ **THEN**
AllTrue = FALSE
ENDIF
ENDFOR
IF AllTrue THEN
$\sigma^n_\bullet(x) = InDep$
ELSE
$\sigma^n_\bullet(x) = \sigma^n_\circ(x)$
ENDIF
ELSE
$\sigma^n_\bullet(x) = \sigma^n_\circ(x)$
ENDIF
ENDFOR
FOREACH $n \in N$ **DO**
IF only one node m such that $(m, n) \in E$ **THEN**
$\sigma^n_\circ = \sigma^m_\bullet$
ELSE IF there are two nodes m_1 and m_2 such that $(m_1, n) \in E$ and $(m_2, n) \in E$ **THEN**
$\sigma^n_\circ = \sigma^{m_1}_\bullet \sqcap \sigma^{m_2}_\bullet$
ENDIF
ENDFOR
ENDWHILE
// Step 2:
FOREACH predicate b of any loop **DO**
FOREACH $n \in Infl(b)$ **DO**
IF $(x \in Def(n))$ && $(\sigma^n_\circ(x) \neq LCtrl)$ **THEN**
$\sigma^n_\circ(x) = LCtrl$
ENDIF
ENDFOR
ENDFOR
IF the state of any variable is changed **THEN**
GOTO Step 1
ENDIF

FIGURE 2. The algorithm for determining input-dependent variables.

(ii) For each node n, if there is only one node m such that $(m, n) \in E$, then $\sigma^n_\circ = \sigma^m_\bullet$; if there are two nodes m_1 and m_2 such that $(m_1, n) \in E$ and $(m_2, n) \in E$, then $\sigma^n_\circ = \sigma^{m_1}_\bullet \sqcap \sigma^{m_2}_\bullet$, i.e.

$$\forall x \in V, \sigma^n_\circ(x) = (\sigma^{m_1}_\bullet \sqcap \sigma^{m_2}_\bullet)(x) = \sigma^{m_1}_\bullet(x) \sqcap \sigma^{m_2}_\bullet(x)$$

where $\sqcap$ is the infimum operator defined by $\sqsubseteq$ such that:

$$\forall a \in SV, b \in SV, a \sqcap b = \begin{cases} a, & \text{if } a \sqsubseteq b, \\ b, & \text{if } b \sqsubseteq a. \end{cases}$$

The iteration of the first step consists of the above operations (i) and (ii). Since the ordering is defined on the specific value set, the iteration will eventually be stablized and terminate. The reason is similar to that in traditional dataflow analysis framework [8].

After the iteration is stabilized, if there exists $n \in N$ such that $x \in Ref(n)$ and $x = Undef$, then x is referenced without definition. This is often the case that the program has an error.

In the second step, we will remove the variables that are influenced by a loop structure from the set of input-dependent variables.

For any predicate b of a loop, and for each statement $n \in Infl(b)$, if $x \in Def(n)$ and $\sigma^n_\circ(x) \neq LCtrl$, then update $\sigma^n_\circ(x)$ to $LCtrl$. As a consequence, all the variables that are defined in a loop will be loop-controlled variables and their pre-states will be $LCtrl$.

If the state of any variable at any position is changed in the second step, the control flow will 'goto' the first step and do the checking process of the second step again until nothing is changed any more.

Because the size of set of input-dependent variables at each node decreases monotonously as defined by the ordering $\sqsubseteq$, the iteration of the first and the second steps will eventually terminate.

By enumerating all the loop constructs of the language, it is easy to prove that a normal loop predicate will never be input dependent. For instance,

```
int wLoop(int lupStart, int lupEnd) {
1. while (lupStart< lupEnd){
2.   ...
3.   lupStart=lupStart+1;}
```

As `lupStart` is defined in a loop, the post-state of `lupStart` of statement 3, $\sigma^3_\bullet(lupStart)$, will be $LCtrl$. There are two flows to statement 1, so we have:

$$\sigma^1_\circ(lupStart) = \sigma^{entry}_\bullet(lupStart) \sqcap \sigma^3_\bullet(lupStart).$$

As a result, the $\sigma^1_\circ$ ($lupStart$) will always be $LCtrl$.

3.2. Slicing input-dependent predicates

For a program M, let $PreSet$ be the set of input-dependent predicates. On the basis of the result of Section 3.1, we can easily derive the set of nodes S_{PreSet}, which are relevant to $PreSet$,

$$S_{PreSet} = \{n \mid \forall x \in (Def(n) \cup Ref(n)), \sigma^n_\bullet(x) = InDep\}.$$

As mentioned above, there will be no loop predicate in S_{PreSet}.

Apparently, the resulting set of merging the slices for the nodes in S_{PreSet} will be S_{PreSet} itself. By adding an empty statement as a branch node in S_{PreSet} for the predicates whose branch nodes are not in S_{PreSet}, the new formed S_{PreSet} (it is still referred to as S_{PreSet}) will be a closed up program.

As an example, we consider the subprogram in Fig. 3, which is partially adapted from [5, 6] and rewritten in C. Part of the iteration results for code segment #2 in Fig. 3 are listed in Table 1. Obviously, only the predicate in line 10 is not an input-dependent predicate.

In Table 1, I is the abbreviation of $InDep$, and we omit the variables whose states are $Undef$.

The mode-relevant slice S_{PreSet} is

$$S_{PreSet} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12', 13, 14'\},$$

where $12'$ and $14'$ mean empty statements in lines 12 and 14.

Slicing the example code segment #2 in Fig. 3 will eliminate statements 10, 12, 14, 15.

3.3. Finding all the paths in a mode-relevant slice

Since there is no loop in the mode-relevant slice M' of a program M, it is possible to list all the paths of M' as $P_1, \ldots, P_N$ by applying the algorithm presented in [9], where N is the

```
int GetMode(int X, int Y, int Z) {
  1.    U = (X - Y)*2;
  2.    if (X > Y)
  3.       W = U;
  4.    else W = Y;
  5.    if ((W + Z) > 100) {
  6.       X = X - 2;
  7.       Y = Y + W; }
  8.    else if (X + Z == 100)
  9.       Y = X*Z + 1;
  10.   for (i = 0;i < 99;i++)
  11.      if (X + Y == 0)
  12.         Z = Z + X;
  13.      else
  14          Z = Z + Y;
  15.   return Z; }
```

FIGURE 3. Example code segment #2.

TABLE 1. Part of the iteration results.

Node	$\sigma^n_\circ$	$\sigma^n_\bullet$
1	$X\rightarrow I;Y\rightarrow I;Z\rightarrow I$	$X\rightarrow I;Y\rightarrow I;Z\rightarrow I;U\rightarrow I$
2	$X\rightarrow I;Y\rightarrow I;Z\rightarrow I;U\rightarrow I$	$X\rightarrow I;Y\rightarrow I;Z\rightarrow I;U\rightarrow I$
3	$X\rightarrow I;Y\rightarrow I;Z\rightarrow I;U\rightarrow I$	$X\rightarrow I;Y\rightarrow I;Z\rightarrow I;U\rightarrow I;$ $W\rightarrow I$
10	$X\rightarrow I;Y\rightarrow I;Z\rightarrow I;U\rightarrow I;$ $W\rightarrow I$	$X\rightarrow I;Y\rightarrow I;Z\rightarrow I;U\rightarrow I;$ $W\rightarrow I$
12	$X\rightarrow I;Y\rightarrow I;$ $Z\rightarrow LCtrl;$ $U\rightarrow I;W\rightarrow I$	$X\rightarrow I;Y\rightarrow I;Z\rightarrow LCtrl;$ $U\rightarrow I;W\rightarrow I$

number of paths in M'. Furthermore, each path P_i of M' will be represented as

$$P_i = <i_1, i_2, \ldots, i_{m_i}>, \quad i = 1, \ldots, N,$$

where i_1 is the unique entry node *entry*, i_{m_i} the unique exit node *exit* and m_i the number of nodes in path P_i.

There are 12 paths in the mode-relevant slice of the example code segment #2 in Fig. 3, some of them are

$$P_1 = <entry, 1, 2, 3, 5, 6, 7, 11, 12', exit>,$$
$$P_2 = <entry, 1, 2, 4, 5, 6, 7, 11, 12', exit>,$$
$$P_3 = <entry, 1, 2, 3, 5, 8, 9, 11, 12', exit> .$$

According to the theory of slicing [3, 4], a path that is infeasible in M' must correspond to a path that is also infeasible in M. However, a feasible path in M' may correspond to an infeasible path in M. In the following section, we will find each feasible path in M', which corresponds to a mode.

4. GENERATING PRECONDITION OF MODE

Given a program M and its model-relevant slice M', each path in M' either corresponds to a mode of M or is an infeasible path. In this section, given a path P of M', we either derive its precondition or conclude that it is infeasible. We assume that all the predicates in the programs considered in the paper are linear.

There are many methods to check the feasibility of a path in a program. We apply the methods in [6], which is an improvement of [5], to our framework. On the basis of an arbitrarily chosen initial input I_0 of program M, which can be applied to all the paths of M', the generating process consists of three steps.

(i) Deriving linear arithmetic representation of the predicate functions.
(ii) Constructing a linear constraint system.
(iii) Solving the linear constraint system.

4.1. Deriving linear arithmetic representation of predicate functions

In this section, we construct a linear arithmetic representation for the predicate function corresponding to each branch predicate on path P. For each branch predicate on P, we first formulate a general linear function of all the input variables. Here, we assume that the predicate functions for a given path are linear functions of input variables.

Given a branch predicate node b and its predicate function F, let input $X = <x_1, x_2, \ldots, x_m>$, we call $L(n, X, P) = d_1x_1 + \cdots + d_mx_m + c$ a *linear arithmetic representation* of X on P at b, where $d_s = D(n, F, v_s, P)$, $c = R(i, X, P) - (d_1x_1 + \cdots + d_mx_m)$. $D(n, F, v_s, P)$ is the derivative of predicate function F of node n on path P for input variable v_s, $s \in \{1, \ldots, m\}$.

For example, the linear formulations for the predicate functions corresponding to the branch predicates on path P_1 of the sliced code segment in Fig. 3 are as follows:

$$\begin{aligned} LF_2 &: a_1X + b_1Y + c_1Z + d_1, \\ LF_5 &: a_2X + b_2Y + c_2Z + d_2, \\ LF_{10} &: a_3X + b_3Y + c_3Z + d_3. \end{aligned} \tag{1}$$

The coefficients of the input variables in the above linear functions represent the slopes of the ith predicate function with respect to input variables, respectively. We will calculate these slopes with respective divided differences.

To compute the slope coefficient with respect to a variable, we execute all the input and assignment statements before n_i along path P and evaluate the predicate function at the initial input $I_0 = (i_1, \ldots, i_j, \ldots, i_m)$ and at $I_0 + (0, \ldots, \Delta i_j, \ldots, 0)$, where m is the number of input variables. Then we compute the divided differences:

$$(F(I_0 + (0, \ldots, \Delta i_j, \ldots, 0)) - F(I_0))/\Delta i_j.$$

This gives the value of the coefficient of x_j in the linear function for the predicate function F corresponding to node n_i in P. Similarly, we compute the other slope coefficients in the linear function.

For the code segment in Fig. 3, let $I_0 = (1, 2, 3)$ and let $\Delta X = 1$, $\Delta Y = 1$, $\Delta Z = 1$. We have: $I_0 + (\Delta X, 0, 0) = (2, 2, 3)$, $I_0 + (0, \Delta Y, 0) = (1, 3, 3)$ and $I_0 + (0, 0, \Delta Z) = (1, 2, 4)$. Let F_i represents the predicate function of the ith statement, then $F_2 = X - Y$, $F_5 = W + Z - 100$ and $F_{10} = X + Y$. The coefficient a_2 of the linear function LF_5 can be calculated as below.

$$\begin{aligned} a_2 &= \frac{(F_5(X = 2, Y = 2, Z = 3) - F_5(X = 1, Y = 2, Z = 3))}{\Delta X} \\ &= \frac{(-97 - (-99))}{1} \\ &= 2. \end{aligned}$$

In the same way, we obtain $b_2 = -2$ and $c_2 = 1$.

To compute the constant term d_i, we compute the predicate residual of the predicate it corresponds to. The *predicate residual* $R(n, I, P)$ of a branch predicate of the statement n for an input I is the value of the corresponding predicate function computed by executing the input and assignment statements before n along path P at the input I. We substitute the linear function with the value of input variables in I_0 and the slope coefficients found above, and let it equal to the value of the predicate residual at I_0 computed above. This gives a linear equation in one unknown term and the value of the constant term can be solved.

For the example above, we have

$$d_2 = R(6, I_0, P_1) - (2^*1 - 2^*2 + 1^*3) = (-99) - 1 = -100.$$

By this method, we also obtain: $a_1 = 1$, $b_1 = -1$, $c_1 = 0$, $d_1 = 0$, $a_3 = 3$, $b_3 = -1$, $c_3 = 0$, $d_3 = -2$.

4.2. Constructing linear constraint systems

We construct linear constraints based on the predicates on the given path, using the linear representations computed above. We convert the linear arithmetic representations of predicate functions into a set of inequalities or equalities. If a branch predicate should evaluate to *True* for the given path, the corresponding predicate function is converted into an inequality/equality with the same relational operator as in the branch predicate. On the other hand, if a branch predicate should evaluate to *False* for the given path, the corresponding predicate is converted into an inequality with reversal of the relational operator used in the branch predicate. If a branch predicate has relational operator $\neq$ and should evaluate to *True* for the given path to be traversed, we transform this inequality into its equivalent form $(Exp_1 - Exp_2 > 0) \vee (Exp_1 - Exp_2 < 0)$ [5].

For the path P_1 of the sliced example in Fig. 3, we obtain

$$\begin{aligned} &X - Y > 0, \\ &2X - 2Y + Z - 100 > 0, \\ &3X - Y - 2 = 0. \end{aligned} \tag{2}$$

4.3. Solving linear constraint systems

We propose to solve the linear constraint system directly using a linear programming solver [6]. By defining the target function to be the addition of all the input variables and minimizing the target function, we obtain a linear programming problem. For instance, for the constraints of path P_1 of the sliced example in Fig. 3, we obtain

$$\begin{aligned} &\min X + Y + Z, \\ &\text{satisfying :} \\ &X - Y > 0, \\ &2X - 2Y + Z - 100 > 0, \\ &3X - Y - 2 = 0. \end{aligned} \tag{3}$$

Using the linear programming solver *lp_solve* (freely available from ftp://ftp.es.ele.tue.nl/pub/lp_solve), we obtain a solution $<X = -50, Y = -51, Z = 100>$. This means that path P_1 of the sliced example in Fig. 3 is feasible and it is indeed a mode of the program M, and furthermore the precondition of the mode is specified by the constraint (2). If the solver detects contradictions in the constraint (3), it will conclude that path P_1 is infeasible.

The method proposed in [6] has been proved to be equal to that proposed in [5], so Theorem 1 [5] is also valid to our method.

THEOREM 1. *If the functions of input computed by all the predicate functions for a path are linear, then either the desired program input for the traversal of the path is obtained directly or the path is guaranteed to be infeasible.*

Theorem 1 shows that, for a given linear path, our method can either obtain the precondition of the mode directly or conclude that the path is infeasible.

5. WCET ANALYSIS FOR A GIVEN MODE

After determining the modes of a program M, the WCET estimate will be calculated for each mode to complete the whole task of WCET analysis. Bernat and Burns [2] and Chapman *et al.* [10] present a method of WCET calculation for annotated modes on a simple CISC processor. Here, we present a new WCET analysis method under a given mode both for modern RISC processors and CISC processors.

In the following sections, we present the framework of mode-based WCET analysis in Section 5.1, the principle of WCET analysis for a given mode in Section 5.2 and its applications to modern RISC processors with pipelines and caches in Section 5.3 and to CISC processors in Section 5.4.

5.1. Mode-based WCET analysis framework

The framework of mode-based WCET analysis is depicted in Fig. 4. Besides compilation, it consists of three parts: *program analysis, processor characteristic analysis* and *WCET calculation for a mode*. Program analysis (*high-level analysis*)

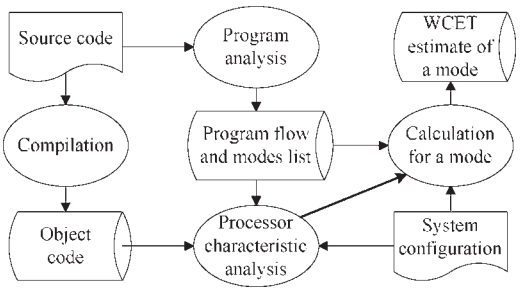

FIGURE 4. Framework of mode-based WCET analysis.

derives information about the possible program flows (such as loop range) and the modes of a program from its source code. Processor characteristic analysis (*low-level analysis*) decides the execution time for atomic parts of the code based on the performance model of the target architecture. The effect of pipelining and caching is considered in low-level analysis for RISC processors. The low-level analysis will make a more precise analysis if it takes the program flow information into account and analyzes individually for each mode. Calculation for a mode takes the information of the two analyses together for each mode in order to derive the exact WCET estimates.

5.2. Principle of WCET analysis for a given mode

Let M' be the mode-relevant slice of program M, and *md* be a mode of M. As mentioned before, there will be a path P' in M' that corresponds to *md* and for each predicate b' in M' there exists a predicate b in M such that b is just the same as b'. Here, we call these predicates as the predicates of mode *md*. A path P in M is *dominated by* a mode *md* if (i) all the predicates of *md* appear in the path P in the same order of the predicates in P'; (ii) the predicates of *md* in P and P' evaluate to the same values. When a path P is dominated by a mode *md*, and $|P| \geq 1$, then the node/basic block on P is said to be dominated by mode *md*. More generally, a node/basic block of M is dominated by mode *md* if there is a path P of M that is dominated by *md*.

For each predicate b of *md*, there are two sets of statements, which are control dependent on b. One is *true* control dependent on b, which is executed under the condition of b evaluating to *true*. The other is *false* control dependent on b under the condition of b evaluating to *false*.

Each predicate b of *md* should take (evaluate to) a value (*true*/*false*) to make it traverse the path P' in M'. If b takes the value *true*, then the nodes that are *true* control dependent on b are dominated by *md* and the nodes that are *false* control dependent on b are not dominated by *md* and vice versa.

For example, for the mode A corresponding to P_1, the predicate $((W + Z) > 100)$ of line 5 in Fig. 3 is on the path of mode A and it should take the value *true*. Therefore, the basic block consisting of lines 6 and 7 in Fig. 3 is dominated by mode A, and the basic block consisting of line 9 is not dominated by mode A.

During WCET analysis of a given mode *md*, we will take the nodes (statements/basic blocks [11]) that is not dominated by *md* out of our consideration. This is the essential reason that the WCET estimates can be obtained more precisely by mode-based approach.

5.3. WCET analysis for RISC processors

The caching analysis [12–14] for WCET exploits traditional data flow analysis framework [8] to obtain the cache state of each instruction. A cache state is simply the subset of all program lines, which can potentially be cached at that point in the control flow [15]. On the basis of its cache state, each instruction is classified as one of four categories: *always hit*, *first hit*, *first miss* and *always miss*.

Let L be the program line that contains an instruction within a basic block. An instruction is categorized as an *always hit* if it is not the first instruction encountered in L in the block, or if L is in the abstract cache state and it does not conflict with any other program line in the same abstract cache state. The instruction is categorized as a *first hit* if the first reference to the instruction will be a hit and all the remaining references during the execution of the loop will be misses. A *first miss* simply indicates that the first reference to the instruction should be treated as a cache miss and all remaining references during the execution of the loop should be considered cache hits. In all other cases, the instruction is conservatively categorized as an *always miss*.

Let L be the program line that contains an instruction within a basic block. The caching analysis for WCET under a given mode *md* is refined as follows: an instruction is categorized as an *always hit* if it is not the first instruction encountered in L in the block, or if L is in the abstract cache state and it does not conflict with any other *md*-dominated program line in the same abstract cache state; as a *first hit* if the first reference to the instruction will be a hit and all remaining references during the execution of the loop will conflict with any other *md*-dominated program line in the same abstract cache state. Otherwise, if an instruction is categorized as *first miss*, it is categorized as *always miss*.

Let n be the maximum number of iterations associated with a loop *lp*, and $Path_{lp}$ be the longest path within loop *lp* that is measured with the WCET estimate [13]. The path-based WCET calculation [12, 13] for loop *lp* can be simplified as

$$WCET_{lp} = n^{*}WCET(Path_{lp}), \tag{4}$$

where $WCET(Path_{lp})$ is the WCET of $Path_{lp}$.

For a given mode *md*, we should restrict the longest path $Path_{lp}$ to be dominated by *md*. If the longest path $Path_{lp}$ within loop *lp* is not dominated by mode *md*, we check the second longest path to see if it is dominated by mode *md*. If the second longest path is dominated by mode *md*, it is used as the longest path in the calculation of formula (4). If the second longest path is not dominated by mode *md*, we continue to check the third longest path within loop *lp*. Because it is apparently that there is at least one path within loop *lp*, which is dominated by mode *md*, we can eventually obtain the longest path $Path_{lp}$, which is dominated by *md*.

5.4. WCET analysis for CISC processors

Generally speaking, CISC processors are less complex than modern RISC processors that usually have pipelines and (multi-level) caches. So it is easier to calculate the numerical WCET of a program for CISC processors [1] according to Section 5.2. For a given mode *md* of a program *M*, we only consider the nodes that are dominated by *md* when computing the WCET of *M* for mode *md*, and the method is simply the same as the existing techniques, such as [16, 17].

6. SYMBOLIC WCET ANALYSIS FOR CISC PROCESSORS

In this section, we present a method to calculate the symbolic WCET of a program for CISC processors based on the method presented in [18]. We assume that the WCET of each basic block is constant and it can be determined at compile time. In other words, it is assumed that the target hardware platform is the processors with fixed instruction execution time, or the processors whose instruction execution time depends on the operands. For instance, although the multiplication and division instruction execution time of some micro instruction processors may vary according to the operands provided, the WCET is constant. Therefore, each node in CFG corresponds to a WCET, which is a constant. We use τ_n to denote the WCET of node n, and use $\tau_{n_1}, \ldots, n_k$ to denote the sum of the WCET values of nodes $n_1, \ldots, n_1$.

6.1. WCET calculation based on branch execution frequencies

This section discusses how to obtain the execution frequencies of symbolic branches using static analysis and how to calculate the WCET values based on branch execution frequencies.

The *execution frequency* $p(m, n)$ of an edge (m, n) of the CFG is a real number satisfying

$$0 \le p(m, n) \le 1, \tag{5}$$

$$\sum_{j \in Succs(m)} p(m, j) = 1, \tag{6}$$

where $Succs(m)$ is the set of all successors of node m. $p(m, n) = 0$ denotes (m, n) infeasible.

The execution frequencies of branches are often denoted by numerical values. If we use algebraic expressions to denote the execution frequencies of the branches, we will obtain the symbolic representation. Given the conditional recursion relationship of the programs, the symbolic formulas of the basic block execution frequencies can be calculated using symbolic instrumentation [18].

We define a symbolic integer variable b_i for each node i. The initial value of b_i is 0. On each access to node i, b_i increases by 1. The calculated symbolic value of b_i is called the *execution count* of node i. b_i is called an *instrumentation variable*.

If we can obtain the symbolic expressions of all the instrumentation variables, then the following equations can be established.

$$b_i = \sum_{j \in Succs(i)} c_{ij}, \tag{7}$$

$$b_i = \sum_{k \in Preds(i)} c_{ki}, \tag{8}$$

where c_{ij} is the symbolic count assigned to edge (i, j), and $Preds(i)$ and $Succs(i)$ are the sets of predecessors and successors of node i, respectively.

According to (7) and (8), we can obtain the expression for c_{ij} using substitution and reduction, and obtain the symbolic execution count for the corresponding edges by back substitution. After knowing the symbolic execution counts of the edges, by (5) and (6) we have

$$p(i, j) = \frac{c_{ij}}{\sum_{k \in Succs(i)} c_{ik}}. \tag{9}$$

With the execution frequencies of each edges, the generating function $G_m(z)$ of node m for variable z is defined as follows [18].

$$\begin{cases} G_m(z) = z^{\tau_m} \sum_{j \in Preds(m)} p(j, m) G_j(z), \\ G_{entry}(z) = z^{\tau_{entry}}. \end{cases} \tag{10}$$

As known in [19], the $WCET_{Task}$ is

$$WCET_{Task} = \frac{d}{dz} G_{exit}(z)|_{z=1}, \tag{11}$$

which means that we differentiate $G_{exit}(z)$ with respect to z and then set $z = 1$.

6.2. WCET analysis for programs containing input-dependent branches

For loop-controlled branches (which corresponds to a loop in program), we can determine their execution frequencies using simple nested heuristics [20] or static analysis [21–23]. However, for non-loop-controlled branches, it is difficult to determine their execution frequencies. Blieberger [18] gives a method to calculate the execution frequencies of loop-controlled branches. Here, we make extension to [18] such that it can be used to the programs, which contain input-dependent branches. It is assumed that the execution frequency of each loop-controlled branch is known, and it is

either a numerical value or a symbolic expression, and it satisfies (5)–(8).

Using the methods such as symbolic evaluation, we can ultimately determine the conditional expressions of the input-dependent nodes. These expressions are symbolic expressions that are determined by the input parameters.

For a branching node depending on inputs, its evaluation behavior does not depend on which loop it is in. Therefore, when the loop is executing, such a branch is either executed or not executed, i.e. its execution frequency is either 1 or 0. Thus, the WCET of the programs that contain input-dependent branches is also determined by the input parameters.

THEOREM 2. *Assuming that node m is an input-dependent branching node and its execution count is Cnt, m has two input-dependent branches* (m, n_1) *and* (m, n_2). *Let*

$$p(m, n_i) = \begin{cases} 1, & \text{if } Cond(m, n_i) = \text{TRUE}, \\ 0, & \text{otherwise.} \end{cases}$$

Then the execution count of node n_i *is* $p(m, n_i)$*Cnt, *where* $p(m, n_i)$ *is the execution frequency of branch* (m, n_i), $i = 1, 2$.

Proof. First, we define the recursive counterpart of a variable. Let v be a variable. $v(k)$ is called a *recursive counterpart* of v, where k denotes the kth occurrence of v in the loop execution.

According to [18], there exists the following recursion relation on the instrumentation variable b_m of node m:

$$b_m(k+1) = e(k), \quad \text{if } C_m(k) \text{ evaluates to TRUE},$$

where $e(k)$ and $C_m(k)$ mean that all the variables in e and C are substituted by their recursive counterparts, and e is the symbolic expression of b_m, and C_m is the condition.

Clearly, $b_{n_i}(k+1) = e(k)$ if $C_m(k)$ evaluates to TRUE, and $Cond(m, n_i) = \text{TRUE}$.

Thus, $b_{n_i}(k+1) = p(m, n_i)*e(k)$ if $C_m(k)$ evaluates to TRUE.

Because the execution count of node m is Cnt, the execution count of node n_i is $p(m, n_i)*Cnt$.

Notice that

$$\begin{cases} 0 \le p(m, n_i) \le 1, & i = 1, 2, \\ \sum_{i=1}^{2} p(m, n_i) = 1. \end{cases}$$

It follows that $p(m, n_i)$ is the execution frequency of branch (m, n_i). □

If the execution frequency $p(m, n_i)$ is given for input-dependent branch, we can still obtain Equation (11) according to the method in Section 6.1. In this case, Equation (11) is the WCET formula that contains the execution frequencies of input-dependent nodes, where $p(m, n_i)$ is a symbol which is either 1 or 0 that denotes the corresponding evaluation of branching node m, respectively. For a better description, we make a sequential numbered list of the execution frequencies $p(m, n_i)$ of all the input-dependent branches accordingly, and make such a convention: if $p(m, n_1)$ is on the TRUE branch of branching node m, and it is labeled p_j, then $p(m, n_2)$ at the FALSE branch is labeled $1 - p_j$. Moreover, for convenience, we call all the p_j and $1 - p_j$ collectively as p. In this way, Equation (11) is a symbolic formula that contains all the p's.

Assume that all the branching nodes of all p's of program M lie in the mode-relevant slice M'. Sections 3 and 4 show that each feasible path in M' corresponds to a mode. For each feasible path in M', there is a value for each p. With the values of all the p's, Equation (11) can be reduced and simplified. According to the value of each p or straightforwardly according to the results in Section 4, we can obtain the preconditions of each mode.

6.3. An example

Figure 5 is a schematic example program adapted from [24]. Figure 6 is the control flow graph of this program. The notes on the right-hand side of the program show the correspondence between the source program statements and the control flow graph. The entry node of Fig. 6 is denoted by a double blank circle, whereas the exit node is denoted by a double filled circle.

6.3.1. *Computing the execution counts of the nodes and the execution frequencies of the edges*

The execution counts of the various nodes and the execution frequencies of the various edges of this program are listed in

```
bool example(integer n, x, cnd1, cnd2) {
1.  i = 0;                  ----Node 1
2.  while (i < n) {          ----Node 2
3.    i++;                  ----Node 3
4.    if (cnd1 > 4) {        ----Node 3
5.      if (i < n-x)         ----Node 4
6.        S1;               ----Node 5
7.      else if (cnd2 = 1)  ----Node 6
8.        S2;               ----Node 7
9.      else S3;             ----Node 8
10  }
11.   if (cnd1 < 3)          ----Node 9
12.     S6;                 ----Node 10
13.   else S7;              ----Node 11
14.   S8;                   ----Node 12
15. }
16. }
```

FIGURE 5. A schematic example program.

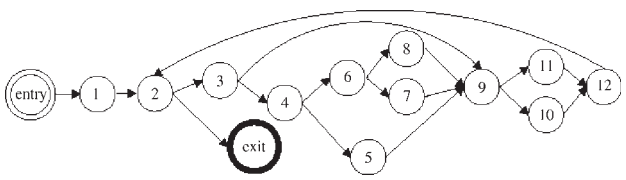

FIGURE 6. Control flow graph for the example program.

TABLE 2. Node execution counts and edge execution frequencies.

Node	Execution count	Edge	Execution frequency
entry	1	(*entry*,1)	1
1	1	(1,2)	1
2	$n+1$	(2,3)	$n/(n+1)$
3	n	(2,*exit*)	$1/(n+1)$
4	p_1*n	(3,4)	p_1
5	p_1*$(n-x)$	(3,9)	$1-p_1$
6	p_1*x	(4,5)	$(n-x)/n$
7	p_1*p_2*x	(4,6)	x/n
8	p_1*$(1-p_2)$*x	(6,7)	p_2
9	n	(6,8)	$1-p_2$
10	p_3*n	(9,10)	p_3
11	$(1-p_3)$*n	(9,11)	$1-p_3$
12	n	(12,2)	1
exit	1		

Table 2. We use p_i to denote the execution frequency of the TRUE branch of the input-dependent node, and $1-p_i$ to denote the other branch. p_i will not appear in the resultant formulae, it is just a means to obtain the WCET formulae and their precondition.

6.3.2. *Computing the generating functions and the symbolic expressions of the WCET of the programs*

The generating function that contains input-dependent branches is given by the following equations.

$$
\begin{aligned}
&G_{entry}(z) = z^{\tau_{entry}}\\
&G_1(z) = z^{\tau_1}\, G_{entry}(z)\\
&G_2(z) = z^{\tau_2}(G_1(z) + G_{12}(z))\\
&G_3(z) = z^{\tau_3}(n/(n+1))G_2(z)\\
&G_4(z) = z^{\tau_4}p_1 G_3(z)\\
&G_5(z) = z^{\tau_5}((n-x)/n)G_4(z)\\
&G_6(z) = z^{\tau_6}(x/n)G_4(z)\\
&G_7(z) = z^{\tau_7}p_2 G_6(z)\\
&G_8(z) = z^{\tau_8}(1-p_2)G_6(z)\\
&G_9(z) = z^{\tau_9}((1-p_1)G_3(z) + G_5(z) + G_7(z) + G_8(z))\\
&G_{10}(z) = z^{\tau_{10}}p_3 G_9(z)\\
&G_{11}(z) = z^{\tau_{11}}(1-p_3)G_9(z)\\
&G_{12}(z) = z^{\tau_{12}}(G_{10}(z) + G_{11}(z))\\
&G_{exit}(z) = z^{\tau_{exit}}(1/(n+1))G_2(z)
\end{aligned}
$$

Since the control flow graph CFG is at most a strongly connected graph, the complexity of computing the generating functions of the nodes of CFG is $O(N^2)$, where N is the number of nodes of CFG.

According to the above equations, we can obtain the following equation after substitution and reduction.

$$G_2 = z^{\tau_{entry,1,2}} + z^{\tau_{3,9,12}}[p_3 z^{\tau_{10}} + (1-p_3)z^{\tau_{11}}]$$
$$[(1-p_1)n + (n-x)p_1 z^{\tau_{4,5}} + xp_1p_2 z^{\tau_{4,6,7}} + xp_1(1-p_2)z^{\tau_{4,6,8}}] \times \frac{G_2}{(n-1)}.$$

After eliminating the recurrence of G_2, we obtain

$$G_2 = \frac{z^{\tau_{entry,1,2}}(n+1)}{(n+1-z^{\tau_{3,9,12}}[p_3 z^{\tau_{10}} + (1-p_3)z^{\tau_{11}}]\Omega)},$$

where

$$\Omega = (1-p_1)n + (n-x)p_1 z^{\tau_{4,5}} + xp_1p_2 z^{\tau_{4,6,7}} + xp_1(1-p_2)z^{\tau_{4,6,8}}.$$

Thus, we have

$$G_{exit} = \frac{z^{\tau_{entry,1,2,exit}}}{(n+1-z^{\tau_{3,9,12}}[p_3 z^{\tau_{10}} + (1-p_3)z^{\tau_{11}}]\Psi)},$$

where

$$\Psi = (1-p_1)n + (n-x)p_1 z^{\tau_{4,5}} + xp_1p_2 z^{\tau_{4,6,7}} + xp_1(1-p_2)z^{\tau_{4,6,8}}.$$

After differentiating on z and then substitute z with 1, we have

$$\frac{d}{dz}G_{exit}|_{z=1} = \Delta + (p_3\tau_{10} + (1-p_3)\tau_{11})n + (n-x)p_1\tau_{4,5} + xp_1p_2\tau_{4,6,7} + xp_1(1-p_2)\tau_{4,6,8},$$

where Δ denotes $\tau_{entry,1,2,3,9,12,exit}$.

6.3.3. *Listing paths and their p combinations*

According to Section 3, after mode-relevant program slicing, we obtain the mode-relevant slice and list all the paths in the slice. For the example in Fig. 5, only statements 4, 7, 9, 11, 13 remain in the mode relevant slice; we list their paths and the corresponding p combinations in Table 3.

6.3.4. *Generating the corresponding conditional expressions*

By applying the method presented in Section 4, we can check whether a path in mode-relevant slice is feasible or not. There are four feasible paths in Table 3. The first and third paths are infeasible.

Table 3. Paths in mode relevant slice and their p combinations.

No.	Paths	p combinations
1	4, 5′, 7, 8′, 11, 12′	p_1, p_2, p_3
2	4, 5′, 7, 8′, 11, 13	$p_1, p_2, 1-p_3$
3	4, 5′, 7, 9, 11, 12′	$p_1, 1-p_2, p_3$
4	4, 5′, 7, 9, 11, 13	$p_1, 1-p_2, 1-p_3$
5	4, 11, 12′	$1-p_1, p_3$
6	4, 11, 13	$1-p_1, 1-p_3$

For the four feasible paths, their individual symbolic expressions and the corresponding conditions are listed below.

$$\begin{cases} \Delta + n\tau_{4,11} + (n-x)\tau_5 + x\tau_{6,7} & \Leftarrow (cnd1 > 4)\ and\ (cnd2 = 1), \\ \Delta + n\tau_{4,11} + (n-x)\tau_5 + x\tau_{6,8} & \Leftarrow (cnd1 > 4)\ and\ (cnd2 \neq 1), \\ \Delta + n\tau_{10} & \Leftarrow (cnd1 < 3), \\ \Delta + n\tau_{11} & \Leftarrow (cnd1 \leq 4)\ and\ (cnd1 \geq 3). \end{cases}$$

Comparing to the unconditional symbolic expression above, it is clear that the symbolic expressions for each mode have been simplified, and their corresponding conditions are orthogonal, i.e. there is no overlapping among their domains.

7. PROTOTYPE TOOL

7.1. Tool architecture

On the basis of our previously developed tools—a path-wise test data generator [25] and a WCET Analyzer [26], we have implemented the presented method and developed a program mode-based WCET automatic analysis tool for C source code. As shown in Fig. 7, our prototype tool is mainly composed of three parts: a parser and a program analyzer, a mode generator and a mode-based WCET analyzer. The mode generator is composed of Mode Relevant Slicer, a Constraint Constructor and a Constraint Solver. The WCET analyzer is composed of Static Cache Monitor and Time Analyzer. The program flow analysis method is based on the value range propagation method [27], which is supported by Abstract Interpretation [28].

A C language *compiler* generates the mapping information between source code and object code. On the basis of this information, the program flow information mentioned above, and the cache configuration information, the *static cache monitor* categorizes each instruction access. *Time analyzer* calculates the execution time of each basic block based on the pipelining information of the instruction, and computes the upper bounds of the processor execution time of program based on the program flow information and the instruction and data caching categorization.

The methods used by static cache monitor and time analyzer are mainly from [12, 13] and are modified for mode.

Time analyzer has been implemented for Alpha 21064, which is a super-scalar and super-pipeline microprocessor with 64-bit load/store RISC architecture and 8 k instruction cache and 8 k data cache in chip [29]. For the simplicity of calculation, we do not take branch prediction and data cache into consideration.

Most parts of the tool execute automatically. All programs flow, but unbounded loops are automatically generated. The tool supports the identification of modes and can determine if a path is dominated by a mode. Static caching monitor and time analyzer can automatically categorize the instruction cache and calculate WCET estimates, respectively. However, the mapping between source code and object code should manually be established by the users at the moment, though it can be established by compiler developers.

7.2. Mode-detection mechanism

We demonstrate mode-detecting process in detail. The flow chart of mode-detecting process is demonstrated in Fig. 8.

The input variable information of the tool includes its name, location (line number) and type in program structure which is generated by a parser. On the basis of this information, mode-relevant slicer generates slices and their paths using the method presented in Section 3. For each path in the slices, constraint constructor generates its constraint system. By invoking Matlab, constraint solver makes decision on whether the path is feasible or not.

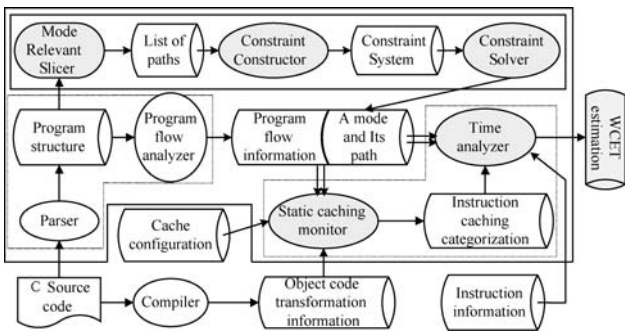

FIGURE 7. Framework of the prototype tool.

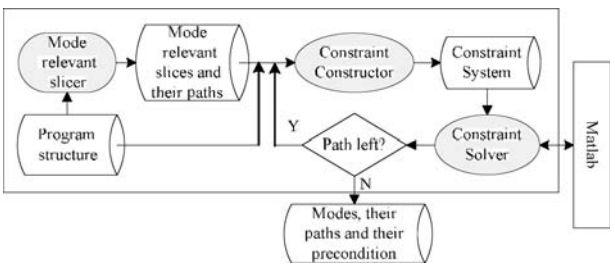

FIGURE 8. Flow chart of mode-detecting process.

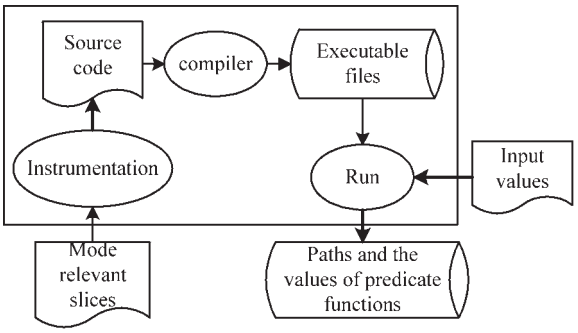

FIGURE 9. Execution process of mode detecting.

Matlab is a mathematical software that integrates calculation and programming. It can resolve several kinds of constraint systems and provide Application Programming Interface (API) for C++ compiler such as the cl.exe of Visual C++. By invoking library functions with C++ API of Matlab, the program of modes based-WCET analysis tool is compiled into an execution file, which is executed stand alone.

According to the input variable information, constraint constructor constructs a linear arithmetic representation for the predicate function corresponding to each branch predicate in the slice, and instruments the slice before and after each branch with assertions. The instrumented slice is compiled and run with various inputs to obtain the executed path and coefficients of the input variables in the linear arithmetic representation as mentioned before. The execution process of mode detecting is depicted in Fig. 9.

8. EXPERIMENTAL RESULTS

With the prototype tool support, we have performed the experiments of worst-case time analysis for RISC processors using SNU-RT Benchmark Suite [30]. There are 17 C programs in the suite, 10 of them contain functions that have modes. In total, there are 57 functions in these 17 programs and 23 of them have modes. Observe that 59% of these programs have modes and that the seven *not-have-mode* programs only contain five functions in total. Table 4 lists all the programs and functions in the SNU-RT Benchmark Suite.

Using our tool and by selecting the initial input value for a program to be analyzed, we can obtain the list of modes of the program, which include the WCET estimate and the precondition for each mode.

For those experiments, Table 5 lists the preconditions and WCET estimates of each mode of some of the functions in the SNU-RT Benchmark Suite that have modes. Each condition in the *Preconditions* column corresponds to a mode. For example, the third function in Table 5 has three preconditions, i.e. ($nbl < 0$), ($0 \leq nbl \leq 18432$) and ($nbl > 18432$). So it has three modes, one for each precondition. Each value in WCET column corresponds to the WCET estimate of the mode. The *Comments* column specifies the conditions under which the WCET estimate is computed. To satisfy this condition, some source codes have to be modified. For instance, some data arrays are enlarged.

TABLE 4. Functions in SNU-RT that have modes.

No.	Program	Functions
1	adpcm-test.c	*abs*, *fabs*, *quantl*, *logscl*, *upzero*, *logsch*
2	crc.c	*icrc*
3	fft1.c	*fabs*, *fft1*
4	fft1k.c	*fabs*
5	fir.c	*fabs*, *sqrt*
6	lms.c	*fabs*, *sqrt*
7	ludcmp.c	*fabs*, *ludcmp*
8	minver.c	*fabs*, *mmul*, *minver*
9	qurt.c	*fabs*, *sqrt*, *qurt*
10	sqrt.c	*sqrt*

By examining the suite, we notice that most mode-relevant predicates are directly represented by input variables.

TABLE 5. Preconditions and WCET estimates of the modes of some functions.

No.	Function	Preconditions	WCET (cycles)	Comment
1	*fabs*	($n \geq 0$)	36	
		($n < 0$)	38	
2	*sqrt*	($val = 0$)	48	
		($val \neq 0$)	2225	
3	*logscl*	($nbl < 0$)	84	
		($0 \leq nbl \leq 18432$)	85	
		($nbl > 18432$)	86	
4	*upzero*	($dlt = 0$)	303	
		($dlt \neq 0$)	493	
5	*icrc*	($jinit \geq 0 \wedge jrev < 0$)	33 364	$len = 100$
		($jinit \geq 0 \wedge jrev \geq 0$)	31 937	
		($jinit < 0 \wedge jrev < 0$)	33 398	
		($jinit < 0 \wedge jrev \geq 0$)	31 956	
6	*fft1*	($n < 2$)	32	
		($n \geq 2 \wedge flag = 0$)	412 702	$n = 16$
		($n \geq 2 \wedge flag \neq 0$)	413 805	
7	*ludcmp*	($n > 99 \vee eps \leq 0.0$)	46	
		($n \leq 99 \wedge eps > 0.0$)	99 013	$n = 10$
8	*mmul*	($row\_a < 1 \vee row\_b < 1 \vee col\_b < 1 \vee col\_a \neq$ row_b)	60	
		($row\_a \geq 1 \wedge row\_b \geq 1 \wedge col\_b \geq 1 \wedge col\_a = row\_b$)	125 433	$row\_a = 10$, $row\_b = 10$, $col\_b = 10$

However, there are still some exceptions, such as the function *icrc* in source file '*crc.c*' of the suite, whose mode-relevant predicates are in loop, and there are even eight infeasible paths in the mode-relevant slice.

On the basis of the WCET estimates of functions listed in Table 5, we obtain the WCET estimates of invoking these functions in programs, as shown in Table 6.

There are five columns in Table 6 except the first column 'No.'. *Function/program* column specifies which function is measured in some program. For example, function *fabs* occurs in programs 1 and 3–9, however we only list its measurements in program 'fir.c' and 'lms.c'.

Column '*WCET without modes*' specifies the WCET estimates of invoking the function while NOT taking mode into consideration. In this case, the WCET estimate is calculated as the maximum estimate of all the modes of the function to be measured.

Column '*WCET with modes*' specifies the WCET estimates of invoking the function when we take mode into consideration. In this case, the WCET estimate is calculated by accumulating each estimate with different mode. For instance, function *upzero* has two modes, one for ($dlt = 0$) and the other for ($dlt \neq 0$). Mode ($dlt = 0$) is executed 2370 times and mode ($dlt \neq 0$) is 1630 times in program *adpcm.c*. Therefore, the WCET with mode for *upzero* is $2370 * 303 + 1630 * 493 = 1\,521\,700$. However, the WCET without mode is $493 * 4000 = 1\,972\,000$.

The 'percentage'column specifies the percentage of the estimate with modes to the estimate without modes. It specifies the accuracy we improved for the functions that have modes. For example, for function *sqrt* in program *qurt.c*, the percent of the estimate with mode to the estimate without mode is $4498/6675 = 67\%$.

Comments column also specifies the conditions under which the WCET estimate is computed.

By examining the programs above, we can find that modes are set for two cases: one for code sharing and another for input variable range checking. For the former, each mode is a 'normal' case, so each mode has a good chance to be taken. Functions 'fabs', '*sqrt*', '*upzero*', '*icrc*' and "*fft1*" are such cases. For the latter, some modes are normal cases, whereas the others are abnormal and they happen unexpectedly. We set the invoking context to be that normal modes happen regularly and abnormal modes happen one-tenth.

The meaning of comment '$n = 0, <=100; +10$' for the sixth program is that we changed the invoking context from original

```
chkerr=ludcmp(nmax, n, eps);
```

to

```
for (n=0;n<=100;n+10)
  chkerr=ludcmp(nmax, n, eps);
```

As we mentioned before, the function *ludcmp* will be called ten times as mode ($n \leq 99$) and once as mode ($n > 99$).

The comment of seventh item in Table 6, 'Param = 1∼10', means all the four parameters of function *mmul* are same and will traverse from 1 to 10. That is, it will be called 10 times.

The comment of the third and eigth items mean the same as that of seventh item.

Some functions that have modes have not been listed in Table 6, such as *quantl*, *logscl* and *logsch* in program *adpcm.c*. The functions are not listed in Table 6 either because they are invoked in the mode that has the maximum estimate of all the modes or because the precondition of each mode is hard to evaluate. For these functions, we take their estimates with mode to the same as without mode.

There are 23 functions in Table 4. As program *sqrt.c* is not an executable program, we do not take function *sqrt* in it into account. Therefore, the average percentage of the estimate with mode to the estimate without modes for this example is $(77 + 96 + 75 + 91 + 90 + 90 + 67 + 9 * 99 + 100 * 6)/22 = 94\%$, i.e. the improvement of our method for this example is 6%. Here, we take all the *fabs* and *abs* functions as 1% improved.

TABLE 6. WCET estimates with modes vs. without modes for some of the programs that have modes.

No.	Function/program	WCET without modes	WCET with modes	Percentage	Comment
1	*upzero*/adpcm.c	1 972 000	1 521 700	77	
2	*icrc*/crc.c	300 582	289 626	96	*jint* = −1, 0, 1 *jrev* = −1, 0, 1
3	fft1.c	6 620 880	4 959 170	75	*n* = 0∼8 flag = 0, 1
4	*fabs*/fir.c	10 792	10 708	99	
5	*fabs*/lms.c	132 772	131 428	99	
6	ludcmp.c	1 089 143	990 176	91	$n = 0, <=100; +10$
7	*mmul*/*minver*.c	1 254 330	1 128 957	90	Param = 1∼10
8	*minver*/*minver*.c	2 013 580	1 812 278	90	Param = 1∼10
9	*sqrt*/qurt.c	6675	4498	67	

In general, the improvement made by our method is dependent on (i) that a function has several modes and the difference between the modes is significant; (ii) how often the mode with minor WCET estimate is invoked.

It is noticed that some precondition is hard to evaluate statically. To evaluate the precondition, the other techniques such as constant propagation [31] should be applied.

9. RELATED WORK AND CONCLUSIONS

The modes of programs in WCET analysis have been investigated in [2, 10]. They annotate programs with modes and compute the WCET value for each given mode using a tree-based method. Our method automatically finds the modes of a program and calculates the WCET estimates under a given mode using a path-based method.

Using a tree-based method, Gheorghita *et al.* [32] have explored the scenario that is defined as the application behavior for a specific type of input data, which is a specific type of mode. Our method is more general and the method of identifying infeasible paths is also more general.

A key point of WCET analysis is to ensure the accuracy of the resulting WCET estimates. Exploiting program modes can make WCET analysis more accurate. In this paper, we present a novel method for deriving modes of a program and calculating the WCET estimate under a given mode automatically. On the basis of program slicing and iterative relaxation method, a general method is presented to obtain the program modes and it works well in practical programs with linear branching predicates. The WCET analysis of a given mode for modern RISC processors with caches and pipelines is simple and can be automated. For non-linear predicates, the constraint systems are not always solvable to obtain its precondition for a mode as linear constraint systems.

Blieberger [18] uses data flow framework to estimate the symbolic WCET of real-time programs. If the conditional recursion relation of the program is already known, then this method can compute the symbolic execution frequencies of CFG nodes using symbolic instrumentation techniques. The generating functions are employed to acquire the accurate WCET symbolic expressions. Note that this method considers only the execution frequencies of loop-controlled branches, which are relatively easy to determine. However, the execution frequencies of non-loop-controlled branches are difficult to determine [33]. Input-dependent branches are just non-loop-controlled branches. On the basis of the Blieberger's work [18], our method can conduct mode-based symbolic WCET analysis for traditional CISC processors. Grounded on the analysis of program modes, we directly generate the WCET symbolic expressions of a program for each mode. However, existing symbolic WCET analysis methods [21, 34] are not targeted at program modes.

As future work, we plan to apply our method to detect program modes for the calculation of energy cost in power-aware computing. We will extend the work in finding approximate preconditions for nonlinear branching predicates and detecting modes for multi-task applications.

FUNDING

This work was partially supported by the National Natural Science Foundation of China under grant no. 60233020, the National High-Tech Research and Development Plan of China under grant no. 2005AA113130 and the Program for New Century Excellent Talents in University under grant no. NCET-04-996.

REFERENCES

[1] Puschner, P. and Burns, A. (2000) Guest editorial: a review of worst-case execution-time analysis. *Real-Time Syst.*, **18**, 115–128.

[2] Bernat, G. and Burns, A. (2000) An approach to symbolic worst-case execution time analysis. *Proc. 25th IFAC Workshop on Real-Time Programming*, Palma, Spain, May 15–19.

[3] Weiser, M. (1984) Program slicing. *IEEE Trans. Softw. Eng.*, **10**, 352–357.

[4] Tip, F. (1995) A survey of prograsm slicing techniques. *J. Program. Lang.*, **3**, 121–189.

[5] Gupta, N., Mathur, A.P. and Soffa, M.L. (1998) Automated test data generation using an iterative relaxation method. *Proc. ACM SIGSOFT 6th Int. Symp. Foundations of Software Engineering*, Orlando, FL, November 3–5, pp. 231–244. ACM Press, New York.

[6] Shan, J.H., Wang, J., Qi, Z.-C. and Wu, J.P. (2003) Improved method to generate path-wise test data. *J. Comput. Sci. Technol.*, **18**, 235–240.

[7] Ferrante, J., Ottenstein, K. and Warren, J. (1987) The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, **9**, 319–349.

[8] Nielson, F., Nielson, H.R. and Hankin, C. (1999) *Principles of Program Analysis*. Springer-Verlag, Berlin.

[9] Baase, S. and van Gelder, A. (2001) *Computer Algorithms: Introduction to Design and Analysis* (3rd edn), pp. 336–356. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

[10] Chapman, R., Burns, A. and Wellings, A. (1994) Integrated program proof and worst-case timing analysis of spark Ada. *Proc. ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-time Systems*, Orlando, FL, June 21, pp. K1–K11, ACM Press, FL.

[11] Aho, A.V., Sethi, R. and Ullman, J.D. (1997) *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

[12] Arnold, R.D., Mueller, F., Whalley, D. and Harmon, M. (1994) Bounding worst-case instruction cache performance. *Proc. 15th*

Real-Time Systems Symposium (RTSS), December 7–9, pp. 172–181, Brookline, MA.

[13] Healy, C.A., Arnold, R.D., Mueller, F., Whalley, D. and Harmon, M.G. (1999) Bounding pipeline and instruction cache performance. *IEEE Trans. Comput.*, **48**, 53–70.

[14] White, R.T., Mueller, F., Healy, C.A., Whalley, D.B. and Harmon, M.G. (1997) Timing analysis for data caches and set-associative caches. *Proc. Real-Time Technology and Applications Symp.*, Montreal, Canada, June 9–11, pp. 192–202. IEEE Computer Society Press, Washington, DC.

[15] Hennessy, J.L. and Patterson, D.A. (1996) *Computer Architecture: A Quantitative Approach* (2nd edn). Morgan Kaufmann Publishers Inc., San Francisco, CA.

[16] Shaw, A.C. (1989) Reasoning about time in high level language software. *IEEE Trans. Softw. Eng.*, **15**, 875–889.

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

[18] Blieberger, J. (2002) Data-flow frameworks for worst-case execution time analysis. *Real-Time Syst.*, **22**, 183–227.

[19] Graham, R.L., Knuth, D.E. and Patashnik, O. (2002) *Concrete Mathematics: A Foundation for Computer Science* (2nd edn), pp. 320–380. Addison-Wesley, NY.

[20] Smith, J.E. (1981) A study of branch prediction strategies. *Proc. 8th Annual Symp. Computer Architecture*, Minneapolis, MN, May 12–14, pp. 135–148. IEEE Computer Society Press, Los Alamitos, CA.

[21] Vivancos, E., Healy, C., Mueller, F. and Whalley, D. (2001) Parametric timing analysis. *Workshop on Languages, Compilers, and Tools for Embedded Systems*, Snowbird, UT, June 22–23, pp. 88–93, ACM Press, New York, NY.

[22] Wu, Y. and Larus, J.R. (1994) Static branch frequency and program profile analysis. *Proc. 27th Int. Symp. Microarchitecture*, San Jose, CA, 30 November–2 December, pp. 1–11. ACM Press, New York, NY.

[23] Patterson, J.R.C. (1995) Accurate static branch prediction by value range propagation. *Proc. ACM SIGPLAN'95 Conf. Programming Language Design and Implementation* (*PLDI'95*), La Jolla, CA, June 18–21, pp. 67–78. ACM Press, New York, NY.

[24] Ji, M.L., Qi, Z.C. and Wang, H.M. (2006) Symbolic WCET analysis of programs containing input-dependent branches. *J. Softw.*, **17**, 628–637.

[25] Shan, J-H., Wang, J., Qi, Z-C., Ma, X. and Shan, L. (2002) Design and implementation of a path-wise automatic generation of test data. *Comput. Eng. Sci.*, **24**, 103–107.

[26] Ji, M.L., Li, J., Wang, X. and Qi, Z.-C. (2006) An automatic WCET analysis tool based on abstract interpretation. *Comput. Eng.*, **18**, 65–70.

[27] Harrison, W.H. (1977) Compiler analysis of the value ranges for variables. *IEEE Trans. Softw. Eng.*, **3**, 243–250.

[28] Cousot, P. and Cousot, R. (1977) Abstract interpretation: a unified model for static analysis of programs by construction or approximation of fixpoints. *Proc. 4th ACM Symp. Principles of Programming Languages*, Los Angeles, CA, January 17–19, pp. 238–252. ACM Press, New York, NY.

[29] Digital Equipment Corporation (1996) Digital semiconductor Alpha 21064 and Alpha 21064A Microprocessors Hardware Reference Manual.

[30] SNU. *SNU real-time benchmarks*. http://archi.snu.ac.kr/realtime/benchmark/.

[31] Killdall, G.A. (1973) A unified approach to global program optimization. *Conf. Recordings of the First ACM Symp. on Principles of Programming Languages*, October, pp. 194–206. ACM Press, New York, NY.

[32] Gheorghita, S.V., Stuijk, S., Basten, T. and Corporaal, H. (2005) Automatic scenario detection for improved WCET estimation. *Proc. 42th ACM/IEEE Design Automation Conference* (*DAC*), Anaheim, CA, June 13–17, pp. 13–17. ACM Press, New York, NY.

[33] Ball, T. and Larus, J. (1996) Efficient path profiling. *Proc. IEEE/ACM Int. Symp. Microarchitecture*, Paris, France, December 2–4, pp. 46–57. IEEE Computer Society Press, Washington, DC.

[34] Lisper, B. (2003) Fully automatic, parametric worst-case execution time analysis. *Proc. 3rd Int. Workshop on Worst-Case Execution Time (WCET) Analysis*, Porto, Portugal, 1 June, pp. 85–88. Mälardalen University, Sweden.