Automated WCET Analysis based on Program Modes

Meng-Luo Ji\(^1\), Ji Wang\(^2\), Shuhao Li\(^2\), and Zhi-Chang Qi\(^1\)
\(^1\) School of Computer, National University of Defense Technology, China
\{jmluo, qzc\}@nudt.edu.cn
\(^2\) National Laboratory for Parallel and Distributed Processing, China
jiwang@mail.edu.cn, shuhao_li@yahoo.com

ABSTRACT
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 of programs. It consists of two phases. In phase one, we firstly automatically find the modes of a program by mode-relevant program slicing; then we compute the precondition for each mode using a path-wise test data generation method; after that, we can either conclude that it is an infeasible path, or get its precondition. In phase two, we calculate the WCET estimate of each given mode for modern RISC processors with caches and pipelines. The experiments are demonstrated to show the effectiveness of the method.

1. INTRODUCTION
Knowing the worst-case execution time (WCET) of tasks is of prime importance for 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 [15]. Precise estimates of WCET can save system resources, so it is desirable for the WCET to be calculated as accurate as possible.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

\(WCET_{pow}(\text{exponent}) = \begin{cases} 1752 - 434*\text{exponent} & \text{exponent < 0} \\ 1474 + 434*\text{exponent} & \text{exponent \geq 0} \end{cases}\)

In this example, the function \(pow\) has two modes, one when \(\text{exponent}\) is negative and the other when \(\text{exponent}\) nonnegative.

When a program has more than one mode, it has different WCET estimates (numbers or parametric formulas) under different modes. Because the preconditions of modes can be quickly evaluated and the decisions can be made when scheduling a task at run-time, or calling a routine under some kinds of context, knowing the WCET
of 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);

The total WCET estimate of calling the function Pow within the body of for statement should be:

\[ \sum_{i=-2}^{8} (1752 + 437 \cdot i) \]

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

\[ \sum_{i=-2}^{8} (1752 - 437 \cdot i) \]

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 for automated obtaining the modes of a program and calculating the WCET estimate under each given mode. It consists of two phases. In phase one, we construct a new program by mode relevant program slicing. We list all the paths of the sliced program. Each path in the sliced program corresponds to a mode or an infeasible path. For each path, by applying the iterative relaxation method [8] [16], we compute the precondition for the path, or prove that the path is infeasible. The precondition of a path is a set of constraints of the input variables of a program that guarantee the program to execute along the path. In phase two, we calculate the WCET estimate of modern RISC processor with caches and pipelines under each given mode.

The rest of the paper is structured as follows. Some preliminary concepts about program analysis are introduced in Section 2. Section 3 describes the mode-relevant program slicing technique. Section 4 proposes an algorithm to generate the precondition for each mode. Section 5 presents a WCET analysis method under each given mode of a program for a simple RISC processor. Section 6 describes our prototype tool and some experiments with the tool. In section 7, we discuss the related works and conclude the paper.

2. PRELIMINARIES

We define a program module \( M \) as a directed control flow graph \( CFG = (N, E, entry, exit) \), where \( N \) is a set of nodes, \( E \) is a set of edges, \( entry \) is a unique entry node and \( exit \) is a unique exit node of \( M \). A node \( n \in N \) represents a single statement or a conditional expression, and 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 \( P=[n_1, n_2, ..., n_k] \in CFG \) is a sequence of nodes such that \( (n_i, n_{i+1}) \in E \) for \( i = 1 \) to \( k \). The length of 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 \( D_i \) of input variable \( i \) is the set of all possible values it may assume. An input vector \( I = (t_1, t_2, ..., t_m) \in (D_1 \times D_2 \times ... \times D_m) \) is called a program input or input, where \( m \) is the number of input variables.

A conditional expression in a multi-way decision statement is called a Branch Predicate (or simply predicate). Here we assume that the branch predicates are simple relational expression (inequalities and equalities) of the form \( E_i \cdot op \cdot E_j \), where \( E_i \) and \( E_j \) are arithmetic expressions, and \( op \) is one of \( \{<, \leq, >, \geq, =, \neq \} \).

Each branch predicate can be transformed into the equivalent branch predicate of the form \( F \cdot op \cdot 0 \), where \( F \) is an arithmetic expression \( E_i \cdot E_j \). 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 \( n \). Node \( n_i \) is flow dependent on node \( n_j \) if there exists a variable \( x \) such that: (1) \( x \in Def(n_i) \), \( x \in Ref(n_j) \), and (2) there exists a path from \( n_i \) to \( n_j \) without intervening definition of \( x \).

A node \( n \) in \( CFG \) is post-dominated by a node \( m \) if all paths from \( n \) to \( exit \) pass through \( m \). A node \( n \) is control dependent on a node \( m \) if (1) there exists a path \( P \) from \( m \) to \( n \) such that for any node \( u \) in \( P \), \( u \in Def(n) \) and \( u \in Def(n) \), \( u \) is post-dominated by \( n \); (2) \( m \) is not post-dominated by \( n \). For programs with structured control flow, the statements in the branches of a predicate \( b \) are control dependent on predicate \( b \). And the nodes in the body of a loop structure are defined as loop controlled nodes.

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

3. MODE RELEVANT PROGRAM SLICING

A mode of a program is to keep the program to execute in a regular pattern or trajectory which is determined by the values of input variables. So we only consider the predicates that are totally dependent on input variables. These predicates are called input dependent predicates. Here we will extract the input dependent predicates relevant statements from the program and form a new program. The new program is called mode-relevant slice.

To detect the mode of a program, we slice the program in a forward way after determining the input dependent variables (a variable which is totally dependent on input variables) of the program by a data-flow framework [14].

3.1 Determining Input Dependent Variables

We define \( \sigma \) to be a function which maps the variables in \( V \) to the values of a specific value set \( SV \). Here \( \sigma \) is called an (abstract) state. For a statement \( n \), we denote the state before \( n \) as \( \sigma_n^0 \) and the state after \( n \) as \( \sigma_{n+1} \). The specific value set \( SV \) is defined as \( SV = [IDep, Undef, LCtrl] \), where a variable is \( IDep \) if it is totally dependent on input variables and a variable is \( LCtrl \) if it is controlled by loop. Initially, all variables are \( Undef \). We define the order: \( LCtrl \subseteq IDep \subseteq Undef \).

Our iteration algorithm consists of two steps.

In the first step, a dataflow framework is used to analyze the program.
(1) Initialization. For node entry, the input variables are assigned IDenp, other variables are assigned Udef.

\[ \sigma_{\text{entry}}^{\text{init}}(x) = \begin{cases} \text{IDenp} & \text{if } x \text{ is an input variable} \\ \text{Udef} & \text{otherwise} \end{cases} \]

For any statement \( n \) other than entry, the variables are assigned Udef.

(2) For any node \( n \), if variable \( x \in \text{Def}(n) \) and \( \sigma^*_x(x) = \text{LCtrl} \) then \( \sigma^*_x(x) = \text{LCtrl} \), else

\[ \sigma^*_x(x) = \begin{cases} \text{IDenp} & \text{if } x \in \text{Def}(n) \text{ and } \forall y \in \text{Ref}(n), \sigma^*_x(y) = \text{IDenp} \\ \sigma^*_x(x) & \text{otherwise} \end{cases} \]

For any node \( n \), if there is only one node \( m \) such that \((m, n) \in E\) and \( (m_1, n) \in E \) and \( (m_2, n) \in E \), then \( \sigma^*_x = \sigma^*_m \cap \sigma^*_n \), i.e., \( \forall x \in V, \sigma^*_x = (\sigma^*_m \cap \sigma^*_n)(x) = \sigma^*_m(x) \cap \sigma^*_n(x) \)

where \( \cap \) is the infimum operator defined by \( \subseteq \).

The iteration of the first step consists of the above operations (2) and (3). The iteration will be eventually stabilized because of the order defined on the specific value set.

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 \( b \) including loop predicates, if \( \exists x \in \text{Ref}(b), \sigma^*_x(x) \neq \text{Idenp} \), then \( b \) is a non-input dependent predicate. For each non-input dependent predicate \( b \) and each statement \( n \in \text{INFL}(b) \), if \( x \in \text{Def}(n) \) and \( \sigma^*_x(x) \neq \text{LCtrl} \), we let \( \sigma^*_x(x) = \text{LCtrl} \).

If the state of any variable is changed in the above processes, we repeat the iteration of (2) and (3) in the first step and do the checking process of the second step again until nothing is changed any more. Because the set of input dependent variables at each node decreases monotonically, the iteration of first step and second step will eventually terminate.

### 3.2 Slicing Input Dependent Predicates

For a program \( M \), let \( \text{PredSet} \) be the set of input dependent predicates. Based on the result of subsection 3.1, we can easily derive the set of statements \( S_{\text{PredSet}} \), which are relevant to \( \text{PredSet} \),

\[ S_{\text{PredSet}} = \{ n | \forall x \in (\text{Def}(n) \cup \text{Ref}(n)), \sigma^*_x(x) = \text{IDenp} \} \]

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

```c
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 2. An example code segment 2.**

<table>
<thead>
<tr>
<th>Node</th>
<th>( \sigma^* )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X\rightarrow Y\rightarrow I,Z\rightarrow I,U\rightarrow I</td>
</tr>
<tr>
<td>2</td>
<td>X\rightarrow I,Y\rightarrow I,Z\rightarrow I,U\rightarrow I</td>
</tr>
<tr>
<td>3</td>
<td>X\rightarrow I,Y\rightarrow I,Z\rightarrow I,U\rightarrow I</td>
</tr>
<tr>
<td>4</td>
<td>X\rightarrow I,Y\rightarrow I,Z\rightarrow I,U\rightarrow I</td>
</tr>
<tr>
<td>5</td>
<td>X\rightarrow I,Y\rightarrow I,Z\rightarrow I,U\rightarrow I</td>
</tr>
</tbody>
</table>

In table 1, \( I \) is the abbreviation of \( \text{IDenp} \). And we omitted the variables whose values are Udef.

The mode-relevant slice \( S_{\text{PredSet}} \)

\[ S_{\text{PredSet}} = \{1,2,3,4,5,6,7,8,9,11,12',13,14'\} \]

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

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

### 3.3 Finding All the Paths in Mode Relevant Slice

For the mode relevant slice \( M \) of a program \( M \), we list all the paths of \( M \) as \( P_1,..., P_N \) by applying the algorithm presented in [3], where \( N \) is the number of paths in \( M \). Furthermore, each path \( P_i \) of \( M \) will be represented as:

\[ P_i = \{ i_1, i_2, ..., i_m \}, i = 1, ..., N \]

where \( i_1 \) is the unique entry node \( \text{entry} \), \( i_n \) is the unique exit node \( \text{exit} \), and \( m \) is number of nodes in path \( P_i \).

There are 12 paths in the mode relevant slice of code segment 2 in figure 2, some of them are:

\[ P_1 = \{ \text{entry}, 1,2,3,5,6,7,11,12',\text{exit} \} \]

\[ P_2 = \{ \text{entry}, 1,2,4,5,6,7,11,12',\text{exit} \} \]

\[ P_3 = \{ \text{entry}, 1,2,3,5,8,9,11,12',\text{exit} \} \]

### 4. GENERATING PRECONDITION OF MODE

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

We apply the methods of [16], which is an improvement of [8], to our framework. Based on 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:

1. Derivation of linear arithmetic representation of the predicate functions;
2. Construction of a linear constraint system;
3. Solving the linear constraint system.

For the code segment 2 in figure 2, we let \( I_0 = (1, 2, 3) \).

### 4.1 Derivation of Linear Arithmetic Representation of Predicate Functions

In this step we construct a linear arithmetic representation for the predicate function corresponding to each branch predicate on \( P \). 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, so it is precisely a representation.

Given a branch predicate node \( b \) and its predicate function \( F \), let input \( X=x_1, x_2, \ldots, x_n > \), we call \( \text{Lin}(n, P) = d_1 x_1 + \ldots + d_n x_n + c \) a linear arithmetic representation of \( X \) on \( P \) at \( b \), where \( d_i = \text{Der}(n, F, v_0, P) \), \( c = \text{R}(n, X, P) = d_1 x_1 + \ldots + d_n x_n \) is the derivative of predicate function \( F \) of node \( n \) on path \( P \) for input variable \( v_0 \).

For example, the linear formulations for the predicate functions corresponding to the branch predicates on path \( P_1 \) of the sliced code segment in figure 2 are:

\[
\begin{align*}
\text{LF}_1: \quad & a_1 X + b_1 Y + c_1 Z + d_1 \\
\text{LF}_2: \quad & a_2 X + b_2 Y + c_2 Z + d_2 \\
\text{LF}_0: \quad & a_3 X + b_3 Y + c_3 Z + d_3
\end{align*}
\]

The coefficients of the input variables in the above linear functions represent the slopes of the \( i \)-th predicate function with respect to input variables, respectively. We approximate 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 \( n \) along path \( P \) and evaluate the predicate function at the initial input \( I_0 = (i_0, \ldots, i_n) \) and at \( I_0 + (0, \ldots, \Delta i_m, 0) \), where \( m \) is the number of input variables. Then we compute the divided differences:

\[
(F(I_0 + (0, \ldots, \Delta i_m, 0)) - F(I_0))/\Delta i_m
\]

This gives the value of the coefficient of \( x_i \) 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 example above, we let \( \Delta X = 1, \Delta Y = 1, \Delta Z = 1 \), then we have:

\[
i_0 + (\Delta X, 0, 0) = (2, 2, 3), i_0 + (0, \Delta Y, 0) = (1, 3, 3), \text{ and } i_0 + (0, 0, \Delta Z) = (1, 2, 4).
\]

Let \( F_1 \) represent the predicate function of \( i_1 \)-th statement, then \( F_1 = X-Y \), \( F_2 = W+Z-100 \), and \( F_3 = X+Y \). The coefficient \( a_i \) of the linear function \( LF_2 \) can be calculated by:

\[
a_i = (F_2(x=2, y=2, z=3) - F_2(x=1, y=2, z=3))/\Delta X = (-97 - (-99))/1 = 2
\]

In the same way, we get \( b_2 = -2 \) and \( c_2 = 1 \).

To compute the constant term \( d_0 \), 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 and it can be solved for the value of the constant term.

For the example above, we have:

\[
d_0 = R(6, I_0, P_1) - (2*1 - 2*1 + 1*3) = (-99) - 1 = -100.
\]

By this method, we also get: \( a_1 = 1, b_1 = -1, c_1 = 0, d_1 = 0, a_2 = 3, b_2 = -1, c_2 = 0, d_2 = -2 \).

### 4.2 Constructing Linear Constraint System

We construct linear constraints based on the predicates on the given path, using the linear representation computed above. We convert the linear arithmetic representations of predicate functions into a set of inequalities and equalities. If a branch predicate should evaluate to \( \text{true} \) 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 \( \text{false} \) 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 \( \text{true} \) for the given path to be traversed, we transform this inequality into its equivalent form \((\text{Exp}_1 \neq \text{Exp}_2 > 0) \lor (\text{Exp}_1 \neq \text{Exp}_2 < 0)\) [8].

For the path \( P_1 \) of the sliced example in figure 2, we get:

\[
\begin{align*}
X - Y & > 0 \\
2X - 2Y + Z - 100 & > 0 \\
3X - Y - 2 & = 0
\end{align*}
\]

### 4.3 Solving Linear Constraint System

We propose to solve the linear constraints system directly by a linear programming solver in [16]. By defining the target function to be the addition of all the input variables and minimize the target function, we form a linear programming problem. For instance, for the constraints of path \( P_1 \) of the sliced example in figure 2, we get:

\[
\begin{align*}
\text{min } X + Y + Z \\
\text{satisfying: } \\
X - Y & > 0 \\
2X - 2Y + Z - 100 & > 0 \\
3X - Y - 2 & = 0
\end{align*}
\]

Using the linear programming solver lp_solve (freely available from ftp://ftp.es.ele.tue.nl/pub/lp_solve), we get a solution \(<X = -50, Y = -51, Z = 100>\). This means that path \( P_1 \) of the sliced example in figure 1 is feasible and it is a mode of the program \( M \), and furthermore the precondition of the mode is the constraint (2).

The method proposed in [16] has been proved to be equal to that proposed in [8], so the following theorem also applies to our method.

**Theorem** [8]: 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.
This theorem shows that, for a given path, our method can prove that the path is infeasible, or obtain the precondition of the path directly.

5. WCET ANALYSIS UNDER A GIVEN MODE
After determining the modes of a program \( M \), to complete the task of WCET analysis, the WCET estimate can be calculated under each mode. However, it is not an easy task to calculate the WCET of a program on various processor architectures under a given mode, and it is doubtful if there is a general WCET analysis method to fulfill such tasks. [4] and [5] have presented 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 for modern RISC processors with pipelines and caches.

5.1 Solving Linear Constraint System
The framework of WCET analysis including modes is depicted in figure 3. It consists of three parts: program analysis, processor characteristic analysis and calculation for a mode. Program analysis (high-level analysis) derives information about the possible program flows such as loop range, and the modes of program from 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 would make a more precise analysis if it takes the program flow information into account and analyzes once for a mode. Calculation for a mode takes the information of the two analyses together once for each mode in order to derive the actual WCET estimates.

5.2 WCET Analysis under 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 (1) all the predicates of \( md \) is on path \( P \) and they keep the order of the predicates in \( P' \); (2) the predicates of \( md \) in \( P \) and \( P' \) evaluate to same value. If \(|P| = 0\), in other words \( P' \) is empty, \( P \) is dominated by any mode. When a path \( P \) is dominated by a mode \( md \), and \(|P| > 1\), then we say the node/basic block on \( P \) is 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 that are control dependent on \( b \). One is true control dependent on \( b \) that 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) that makes 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. Apparently, only these nodes are not dominated by \( md \).

For example, for the mode \( A \) corresponding to \( P_1 \), the predicate \(((W+Z) > 100)\) of line 5 in figure 2 is on the path of mode \( A \) and it should take the value true. So the basic block consisting of line 6 and 7 in figure 2 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 should take the nodes (statements/basic blocks [1]) that is not dominated by \( md \) out of our consideration. Except for this consideration, the WCET analysis methods need not change any more.

5.3 Caching Analysis and WCET Calculation
The caching analysis [2] [11] for WCET exploits traditional data flow analysis framework [14] to get the cache state of each instruction. A cache state is simply the subset of all program lines [12] that can potentially be cached at that point in the control flow. Based on the cache state of each instruction, 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. 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; an instruction is categorized 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 [11]. The path-based WCET calculation [2] [11] for loop \( lp \) can be simplified as:

\[
 WCET_{lp} = n \cdot WCET(Path_{lp})
\]  

(4)

where \( WCET(Path_{lp}) \) is 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 not 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 \( m_d \), we continue to check the third longest path within loop \( l_p \). Because it is apparently that there is at least one path within loop \( l_p \) that is dominated by mode \( m_d \), we can get the longest path \( Path_m \) that is dominated by \( m_d \).

### 6. IMPLEMENTATION AND EXPERIMENTS

#### 6.1 The Prototype Tool

Based on our previously developed tools – a Path-wise Test Data Generator (PTDG) [17] and a WCET Analyzer [13], we realized our method in this paper and developed a program model based WCET automatic analysis tool for C source code. As shown in figure 4, our prototype tool is mainly composed of three parts: a parser, a mode generator and a WCET analyzer under mode. 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 value range propagation method [10] which is supported by Abstract Interpretation [6].

![Figure 4. The framework of the prototype tool.](image)

A C compiler generates the mapping information between source code and object code. Based on this information, the program flow information above, and cache configuration information, Static cache monitor categorizes each instruction access. Time analyzer calculates the execution time of each basic block [1] 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 [2] [11] and are modified for mode.

Time analyzer is implemented for Alpha 21064, which is a superscalar and super-pipeline microprocessor with 64-bit load/store RISC architecture and 8K instruction cache and 8K data cache in chip [7]. For the simplicity of calculation, we do not take branch prediction and data cache into consideration.

Most parts of the tool execute automatically. All program flows but unbounded loops are automatically generated. Modes are automatically identified and a path can be automatically determined if it is dominated by a mode. Static caching monitor automatically categorizes the instruction cache. Time analyzer also automatically calculates WCET estimates. However, the mapping between source code and object code should be manually established by the users for the time being, though it can be automatically established by compiler developers.

#### 6.2 Experiment Results

We present our experimental results of analyzing SNU-RT Benchmark Suite [18] for Worst Case Timing Analysis. There are 17 C programs in the suite, 9 of them contain functions that have modes. Altogether, there are 55 functions in these 17 programs and 23 of them have modes. Observe that most of these programs have modes.

In the experiments, modes can be found and calculated by our tool. Table 2 lists the preconditions and WCET estimates of each mode of some functions in the SNU-RT Benchmark Suite. Each condition in Preconditions column corresponds to a mode. For example, the second function in Table 2 has 3 preconditions, they are \((nbl < 0), (0 \leq nbl \leq 18432)\) and \((nbl > 18432)\). So the second function 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 condition 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.

By examination of the suite, we notice that most mode relevant predicates are directly represented by input variables. 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 8 infeasible paths in the mode relevant slice.

**Table 2. The preconditions and WCET estimates of the modes of some functions**

<table>
<thead>
<tr>
<th>No.</th>
<th>Function</th>
<th>Preconditions</th>
<th>WCET (cycles)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( fabs )</td>
<td>((n \geq 0))</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>((n &lt; 0))</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( sqrt )</td>
<td>((val = 0))</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>((val \neq 0))</td>
<td>2225</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( logsc1 )</td>
<td>((nbl &lt; 0))</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>((0 \leq nbl \leq 18432))</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>((nbl &gt; 18432))</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( upzero )</td>
<td>((dlt = 0))</td>
<td>303</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>((dlt \neq 0))</td>
<td>493</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( icrc )</td>
<td>((jinit \geq 0 \land jrev &lt; 0))</td>
<td>33,364</td>
<td>len=100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((jinit \geq 0 \land jrev \geq 0))</td>
<td>31,937</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>((jinit &lt; 0 \land jrev &lt; 0))</td>
<td>33,398</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>((jinit &lt; 0 \land jrev \geq 0))</td>
<td>31,956</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>( fft1 )</td>
<td>((n &lt; 2))</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>((n \geq 2 \land flag = 0))</td>
<td>412,702</td>
<td>(n=16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((n \geq 2 \land flag \neq 0))</td>
<td>413,805</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>( ludcmp )</td>
<td>((n &gt; 99 \land eps \leq 0.0))</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>((n \leq 99 \land eps &gt; 0.0))</td>
<td>99,013</td>
<td>(n=10)</td>
</tr>
</tbody>
</table>
7. RELATED WORKS AND CONCLUSIONS
The modes of program in WCET analysis have been investigated in [4] [5]. They annotate programs with modes and compute the WCET value for each given mode using a tree-based method. Our method automatically finds the mode of program and calculates the WCET estimates under a given mode using a path-based method.

Also using a tree-based method, [9] has explored the scenario which is defined as the application behavior for a specific type of input data. So the scenario is a specific type of mode. Our method is more general and the method of identifying infeasible paths is also more general.

A main guideline of WCET analysis is to ensure the accuracy of the resulted 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. Based on program slicing and iterative relaxation method, a general method is presented to obtain the program modes and it works well in realistic programs with linear branching predicates. For nonlinear branching predicates, the precondition we calculate for each mode is not guaranteed to be accurate. The WCET analysis of a given mode for modern RISC processors with caches and pipelines is simple and can be automated.

As the 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 accurate preconditions for nonlinear branching predicates and detecting modes for multi-task applications.

8. ACKNOWLEDGMENTS
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-04-0996.

9. REFERENCES