Supporting Timing Analysis by Automatic Bounding of Loop Iterations[†]

Christopher Healy
Computer Science Department, Florida State University, FL, USA

Mikael Sjödin
Department of Computer Systems, Uppsala University, Sweden

Viresh Rustagi
Silicon Spice, Mountain View, CA, USA

David Whalley and Robert van Engelen
Computer Science Department, Florida State University, FL, USA

September 23, 1999

Abstract.
Static timing analyzers, which are used to analyze real-time systems, need to know the minimum and maximum number of iterations associated with each loop in a real-time program so accurate timing predictions can be obtained. This paper describes three complementary methods to support timing analysis by bounding the number of loop iterations. First, an algorithm is presented that determines the minimum and maximum number of iterations of loops with multiple exits. Even when the number of iterations cannot be exactly determined, it is desirable to know the lower and upper iteration bounds. Second, when the number of iterations is dependent on unknown values of variables, the user is asked to provide bounds for these variables. These bounds are used to determine the minimum and maximum number of iterations. Specifying the values of variables is less error prone than specifying the number of loop iterations directly. Finally, a method is given to tightly predict the execution time of inner loops whose number of iterations is dependent on counter variables of outer level loops. This is accomplished by formulating the total number of iterations of a loop in terms of summations and solving the resulting equation. These three methods have been successfully integrated in an existing timing analyzer that predicts the performance for optimized code on a machine that exploits caching and pipelining. The result is tighter timing analysis predictions and less work for the user.

Keywords: Hard Real-Time Systems, Worst-Case Execution Time, Static Analysis

1. Introduction

To be able to predict the Best-Case Execution Times (BCETs) and Worst-Case Execution Times (WCETs) of a program, one must know

[†] Part of this work has be previously published in the *Proceedings of IEEE Real-Time Technology and Applications Symposium*, June 1998, under the title "Bounding Loop Iterations for Timing Analysis".

the number of iterations that can be performed by the loops in the program. Under certain conditions, such as a loop with a single exit, many compilers statically determine the exact number of loop iterations (Benitez and Davidson, 1988). Applications for determining this number include loop unrolling (Hennessy and Patterson, 1996), software pipelining (Lam, 1988), and exploiting parallelism across loop iterations (Stone, 1990). When the number of iterations cannot be exactly determined, it is desirable in a real-time system to know lower and upper bounds on the number of iterations. These bounds can be used by a timing analysis tool to more accurately predict BCETs and WCETs.

Many existing timing analyzers require that a user specify the number of iterations of each loop in the program. This specification may be requested interactively (Park and Shaw, 1991; Li et al., 1995). Thus, each time the timing analyzer is invoked for a program, the bounds for every loop in the program must be specified, which is error prone and tedious for the user. Alternatively, one could specify this information as assertions in the source code to prevent repeated specifications of the same information (Burns et al., 1996; Puschner and Koza, 1989; Kligerman and Stoyenko, 1986).

However, there are still several disadvantages. First, the user is still required to write the assertions. Second, there is no guarantee that the user will specify the correct minimum and maximum number of iterations. This problem may easily occur when a user changes the loop, but forgets to update the corresponding assertion. Also, code generation strategies, such as whether to place instructions for the loop exit condition code at the beginning or end of the loop, may cause the number of loop iterations of the transformed loop to vary by one iteration from the loop in the source code. Finally, compiler optimizations, such as loop unrolling or software pipelining, may affect the number of times a loop iterates. Inhibiting different code generation strategies or compiler optimizations to more easily estimate loop bounds would sacrifice performance, which is quite undesirable.

It would be more appropriate to have the compiler automatically and efficiently determine the bounds for each loop in a program when possible. This paper describes three methods that support timing analysis by bounding the number of loop iterations. First, an algorithm is presented that determines bounds on the number of iterations for loops with multiple exits. Second, support is provided for loops whose number of iterations is dependent on loop-invariant variables. Finally, a method is given to accurately predict the number of iterations for inner loops, whose number of iterations varies depending upon the values of counter variables of enclosing outer loops. All three of these methods are efficiently implemented and result in less work for a user. The last method also results in tighter timing analysis predictions. These methods were implemented by modifying the *vpo* compiler (Benitez and Davidson, 1988) to analyze loops and the information about number of loop iterations is passed to a timing analyzer (Arnold et al., 1994; Healy et al., 1995; White et al., 1997; Healy and Whalley, 1999a) to predict performance. Note that these methods applied in *vpo* could be used in other compilers or on assembly or machine code files as well.

2. Related Work

Previous work to bound the number of loop iterations has used abstract interpretation (Ermedahl and Gustafsson, 1997) and symbolic execution (Lundqvist and Stenström, 1998; Liu and Gomez, 1998) to automatically derive the number of loop iterations. These approaches are quite powerful, but effectively requires simulating all paths of a loop for every loop iteration. Thus, they require significant analysis overhead, which would be undesirable when analyzing long running programs.

Our work on bounding iterations for nested loops was inspired by the work of Sakellariou (Sakellariou, 1997; Sakellariou, 1996). He calculated the total number of iterations for loops that are dependent on counter variables of outer loops in order to obtain better load balance by assigning approximately the same number of loop iterations to each processor. The approach used was to formulate summations representing the number of loop iterations by hand and to interface to a mathematical package off-line to solve the equations. In this paper, we describe a method to automatically calculate the average number of times that a loop will iterate and how to use this information to obtain tighter timing predictions.

3. Bounding Iterations for Loops with Multiple Exits

In this section we present a method to determine a bounded number of iterations for natural loops with multiple exits. The method includes the following steps. (1) First, the conditional branches within the loop that can affect the number of loop iterations are identified. (2) Next, we calculate when each of the identified branches can change its result based on the number of loop iterations performed. (3) Afterwards, the range of loop iterations when each of these branches can be reached is determined. (4) Finally, the minimum and maximum number of iterations for the loop is calculated. These steps are described in the following subsections.

3.1. Identifying the Branches That Can Affect the Number of Loop Iterations

In this subsection some terms are defined to facilitate the presentation of the methods in this paper. A more complete description of these terms can be found elsewhere (Aho et al., 1986). We define the number of loop iterations as the number of times the header is executed once the loop is entered (Arnold et al., 1994). A *basic block* is a sequence of instructions with a single entry point at the beginning and a single exit point at the end. A *natural loop* is a loop with a single entry point. The *header* of a natural loop is the single basic block where the loop is entered. Transitions from within the loop to the header are called *back edges.* Block A *dominates* block B if every path from the initial node of the control flow graph to B has to first go through A. For instance, the header of a natural loop dominates all blocks in the loop. Similarly, block B *postdominates* block A if all control paths from block A to the exit node of the graph contains block B. A block always dominates and postdominates itself.

An *iteration branch* in a loop is a conditional transfer of control, where the choice between the two outgoing transitions can directly or indirectly affect the number of loop iterations. The iteration branches in the loop that can directly affect this number are branches that have (1) a transition to a basic block outside the loop or (2) a transition to the header block of the loop or to a block that is postdominated by the header. Iteration branches that can indirectly affect the number of loop iterations are those branches whose two successors are postdominated by different iteration branches. Figure 1 shows an algorithm to calculate the set of iteration branches I for a loop. The worst-case complexity of the algorithm is $O(B^2 I^2)$, where B is the number of basic blocks in the loop and I is the number of iteration branches. However, we believe that the average complexity would be closer to $O(B)$ since iteration branches that indirectly affect the number of loop iterations are not common, particularly in numerical applications.

Figure 2(a) contains the code for a toy example C function that will be used to illustrate the algorithm for calculating loop iteration bounds for loops with multiple exits. Figure 2(b) depicts the RTLs, representing SPARC assembly instructions, that the *vpo* compiler has generated for this function. Note that the relational operator in the conditional branches are sometimes reversed from the relational expressions in the source code. (No delay slots have been filled in order to simplify the example.) Figure 2(c) explains the RTL notation used. The loop consists of basic blocks 2, 3, 5, 6, 7, and 8. The header of the loop is block 7. The algorithm shown in Figure 1 identifies block 5

```
//Find the iteration branches that can directly affect the number of iterations
$I = \{\}$
FOR each block $B$ in the loop $L$ DO
    IF ($B$ has two successors $S_1$ and $S_2$) THEN
        IF ($S_1 \notin L$) OR ($S_2 \notin L$) OR
            ($S_1 \in$ PostDom(Header($L$))) OR ($S_2 \in$ PostDom(Header($L$))) THEN
             $I = I \cup B$
        END IF
    END IF
END FOR

//Find the iteration branches that can indirectly affect the number of iterations
DO
    FOR each block in $B$ in the loop $L$ DO
        IF ($B$ has two successors $S_1$ and $S_2$) AND ($B \notin I$) THEN
            IF (there exists $J, K \in I$ AND $J \neq K$ AND
                $S_1 \in$ PostDom($J$) AND $S_2 \in$ PostDom($K$)) THEN
                  $I = I \cup B$
            END IF
        END IF
    END FOR
WHILE (any change to $I$)
```

Figure 1. Finding the Set of Iterations Branches for a Loop

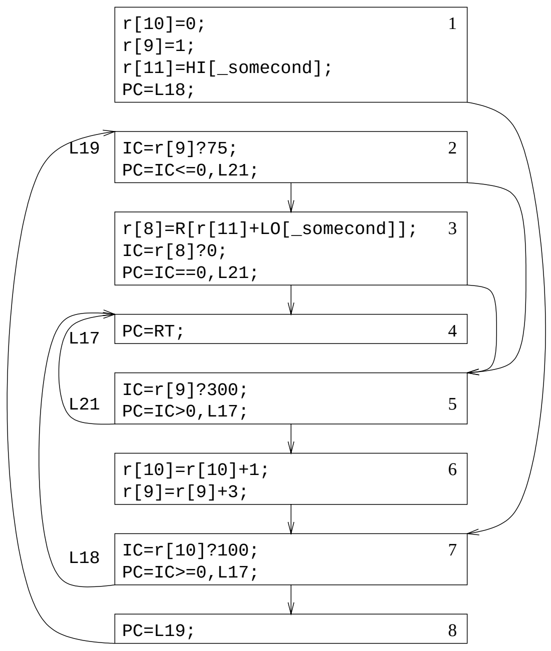
as containing an iteration branch since it has a transition to block 6, which is postdominated by the loop header. Blocks 3, 5, and 7 are identified as having iteration branches since they have a transition to block 4, which is not in the loop. Block 2 is added to the set of blocks containing iteration branches since it can indirectly affect the number of iterations by transferring control to either block 3 or block 5, which both have been identified as containing iteration branches.

Once the blocks containing iteration branches for the loop have been identified, a precedence is established that represents the order that these blocks can be executed on any given iteration of the loop. This precedence relationship can be represented as a Directed Acyclic Graph (DAG). The nodes in the DAG represent the blocks containing the iteration branches and two additional nodes, `continue` and `break`. Figure 3 shows the DAG depicting the precedence relationship between the blocks containing exit conditions from Figure 2. The construction of the DAG can conceptually be accomplished by starting with the graph representing the loop, replacing all back edges with

```
main()
{
   int i, j;
   extern int somecond;

   for (i = 0, j = 1; i < 100; i++, j += 3)
      if (j > 75 && somecond || j > 300)
         break;
}
```

(a) Source Code



```
r[10]=0;                                  1
r[9]=1;
r[11]=HI[_somecond];
PC=L18;

L19  IC=r[9]?75;                          2
     PC=IC<=0,L21;

r[8]=R[r[11]+LO[_somecond]];              3
IC=r[8]?0;
PC=IC==0,L21;

L17  PC=RT;                               4

L21  IC=r[9]?300;                         5
     PC=IC>0,L17;

r[10]=r[10]+1;                            6
r[9]=r[9]+3;

L18  IC=r[10]?100;                        7
     PC=IC>=0,L17;

PC=L19;                                   8
```

(b) Corresponding RTL Instructions

`r[9]`	: register allocated to variable j
`r[10]`	: register allocated to variable i
`HI[<address>]`	: high portion of address
`LO[<address>]`	: low portion of address
`R[<address>]`	: integer memory reference
`IC=<item>?<item>;`	: comparison
`PC=IC<relop>0,<label>;`	: conditional branch
`PC=RT;`	: return
`PC=<label>;`	: unconditional jump

(c) Explanation of RTL Notation

Figure 2. Example Loop with Multiple Exits

transitions to `continue`, replacing each transition out of the loop with a transition to `break`, and collapsing all nodes that do not represent iteration branches. The actual implementation of the DAG construction started with only nodes representing `continue`, `break`, and blocks containing iteration branches and used domination and postdomination information to establish the edges between the nodes. This algorithm

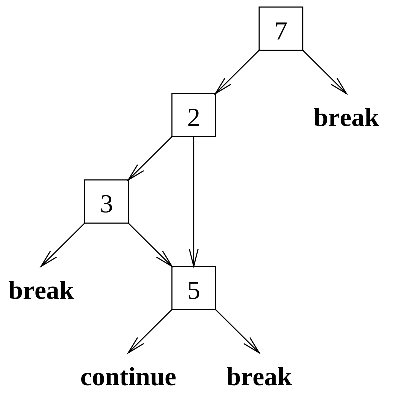


Figure 3. Precedence Relationship between Iteration Branches from Figure 2

is essentially a sort and requires $O(I \log I)$ complexity, where I is the number of iteration branches in the loop.

3.2. Determining When Each Iteration Branch Changes Direction

In this subsection a technique is presented that calculates when each iteration branch can change its result based on the number of loop iterations performed. This technique is similar to those used by other compilers that can calculate the number of iterations of a loop with a single exit (Benitez and Davidson, 1988). For each iteration branch we derive the information shown in Table I. When all of the requirements listed in Table I are satisfied, the iteration branch is classified as *known*. Otherwise, the iteration branch is classified as *unknown*. Note that detection of *unknown* iteration branches in a loop does not mean that the number of iterations of a loop cannot be bounded. Using the derived values, we apply Equation 1 to straightforwardly calculate on which iteration, N_i, that a *known* iteration branch i will change direction. Table II shows the values derived for the example in Figure 2. The iteration branch in block 3 is classified as *unknown* since the *variable* `somecond` is not a basic induction variable. The complexity of this algorithm is $O(I)$, where I is the number of iteration branches, since each iteration branch need only be examined once.

$$N_i = \left\lfloor \frac{limit_i - (initial_i + before_i) + adjust_i}{before_i + after_i} \right\rfloor + 2 \qquad (1)$$

In addition, we have to select a value for *adjust* and checks have to be made in case the iteration branch will always or never be satisfied. Table III shows under what conditions we can use Equation 1 to determine when an iteration branch changes direction. The column "Test

Table I. Information Calculated for Each Iteration Branch

Term	Explanation	Requirement
variable	The control variable on which the branch depends, i.e., the variable being compared in the block containing the iteration branch.	The control variable must be a basic induction variable, that is a variable `v` whose only assignments within the loop are of the form `v := v` $\pm$ `c` where `c` is a constant. In addition, we require that the variable change by a constant integer amount on every loop iteration. We ensure this by checking that each basic block containing an assignment to a basic induction variable dominates all of the blocks containing the tails of the back edge transitions.
limit	The value being compared to the *variable* in the block containing the branch.	The limit must be an integer constant. We will describe how this requirement can be relaxed in Section 4.
relop	The relational operator used to compare the *variable* and the *limit*. I.e., the iteration condition is: "*variable relop limit*".	Our initial description requires that the relational operator be an inequality operator (i.e. `<`, `<=`, `>=`, and `>`). We will describe how to relax this requirement in Section 3.5 to more accurately handle the equality operators (i.e. `==` and `!=`).
initial	The value of the *variable* when the loop is entered.[1]	The initial value must be an integer constant. We will describe how this requirement can be relaxed in Section 4.
before	The amount by which the *variable* is changed before reaching the iteration branch in each iteration.	The amount by which the control variable is incremented or decremented must be an integer constant and these changes must occur on each complete iteration of the loop.[2]
after	The amount by which the *variable* is changed after reaching the iteration branch in each iteration.	The amount by which the control variable is incremented or decremented must be an integer constant and these changes must occur on each complete iteration of the loop.
adjust	An adjustment value of -1, 0, or 1, which compensates for the difference between relational operators (e.g. `<` and `<=`).	

[1]This value is found by searching backwards in the control flow for assignments to *variable*. The search starts with the *preheader*, which is the block outside the loop preceding the loop header.

[2]In other words, the basic blocks containing these changes must dominate every predecessor block of the header that is in the loop.

Table II. Derived Information for Each Iteration Branch in Figure 2

branch	variable	register	limit	relop	initial	before	after	adjust	class	N
block 2	j	r[9]	75	<=	1	0	3	0	known	26
block 3	somecond	r[8]	0	==	N/A	0	0	N/A	unknown	N/A
block 5	j	r[9]	300	>	1	0	3	0	known	101
block 7	i	r[10]	100	>=	0	0	1	-1	known	101

Table III. How to Determine When a Branch with an Inequality Test Changes Direction

Operator	Condition	Test Result	*adjust*
<=	$\mathit{first} \leq \mathit{limit}$ & $\mathit{incr} > 0$	is false on the Nth iteration	0
<=	$\mathit{first} \leq \mathit{limit}$ & $\mathit{incr} \leq 0$	always true	
<=	$\mathit{first} > \mathit{limit}$ & $\mathit{incr} \geq 0$	always false	
<=	$\mathit{first} > \mathit{limit}$ & $\mathit{incr} < 0$	is true on the Nth iteration	1
<	$\mathit{first} < \mathit{limit}$ & $\mathit{incr} > 0$	is false on the Nth iteration	-1
<	$\mathit{first} < \mathit{limit}$ & $\mathit{incr} \leq 0$	always true	
<	$\mathit{first} \geq \mathit{limit}$ & $\mathit{incr} \geq 0$	always false	
<	$\mathit{first} \geq \mathit{limit}$ & $\mathit{incr} < 0$	is true on the Nth iteration	0
>	$\mathit{first} \leq \mathit{limit}$ & $\mathit{incr} > 0$	is true on the Nth iteration	0
>	$\mathit{first} \leq \mathit{limit}$ & $\mathit{incr} \leq 0$	always false	
>	$\mathit{first} > \mathit{limit}$ & $\mathit{incr} \geq 0$	always true	
>	$\mathit{first} > \mathit{limit}$ & $\mathit{incr} < 0$	is false on the Nth iteration	1
>=	$\mathit{first} < \mathit{limit}$ & $\mathit{incr} > 0$	is true on the Nth iteration	-1
>=	$\mathit{first} < \mathit{limit}$ & $\mathit{incr} \leq 0$	always false	
>=	$\mathit{first} \geq \mathit{limit}$ & $\mathit{incr} \geq 0$	always true	
>=	$\mathit{first} \geq \mathit{limit}$ & $\mathit{incr} < 0$	is false on the Nth iteration	0

Where $\mathit{first} = \mathit{initial} + \mathit{before}$, $\mathit{incr} = \mathit{before} + \mathit{after}$,
N is defined in Equation 1, and *adjust* is used in Equation 1.

Result" shows under what conditions we can conclude that the iteration condition will always/never be true and when we should use Equation 1 to determine at which iteration the branch changes direction, as well as, what value to use for *adjust* in Equation 1. Note that when *before+after* = 0 we need not use Equation 1 and thus we do not cause a divide by zero exception.

Figure 4 shows two loops where Equation 1 cannot be applied. Our implementation detects that the loop in Figure 4(a) exits after a single iteration. Recall that the number of iterations is the number of times that the loop header block (i.e. testing $i > 100$ in the example) is executed once the loop is entered (see Section 3.1). The loop in Figure 4(b) is classified as *unbounded* since the loop may never exit depending on how overflow of negative integer values is handled.

```
for(i = 0; i > 100; i++)
   A;
```

(a) Loop That Exits Immediately

```
for(i = 0; i < 100; i--)
   A;
```

(b) Loop That May Never Exit

Figure 4. Two Loops Which are Handled Without Equation 1

3.3. Determining the Iterations When Each Iteration Branch Can Be Reached

The next step is to determine the iterations on which it is possible to execute each node of the DAG. Calculating these ranges requires $O(I)$ complexity, where I is the number of iteration branches. We record this information as a range of iterations and attach a range to each node and edge. The DAG is processed top-down.

The head of the DAG is assigned the range $[1..\infty]$. All other nodes are assigned a range that is the union of the ranges of all incoming edges. The outgoing edges of a node i are assigned ranges using one of the following two rules:

1. If iteration branch i is *known*, then $relop_i$ and the direction of the increment (i.e. the sign of $before_i + after_i$) is used to determine which edge is taken on the first $N_i - 1$ iterations. That edge is assigned the range that is the intersection of $[1..N_i - 1]$ and the range of node i. The other outgoing edge is assigned the range that is the intersection of $[N_i..\infty]$ and the range of node i. If a range assigned to an outgoing edge is empty, then this edge corresponds to an infeasible transition and is deleted from the DAG.

2. If iteration branch i is *unknown*, then both outgoing edges are assigned the same range as node i.

Figure 5 shows the DAG of iteration branches in Figure 3 on page 7 with the range of possible iterations for each node and edge also depicted. Nodes with *known* iteration branches are marked with a **K** and *unknown* iteration branches are marked with a **U**. Iteration branch 7 will take the transition to branch 2 on the first 100 iterations. Note this iteration range of [1..100] corresponds to the variable `i`'s value range of [0..99]. At this point, all values of variables have been abstracted as ranges of loop iterations. Node 5's transition to a *break* is deleted since the range associated with that transition is empty (i.e. the transition is not possible).

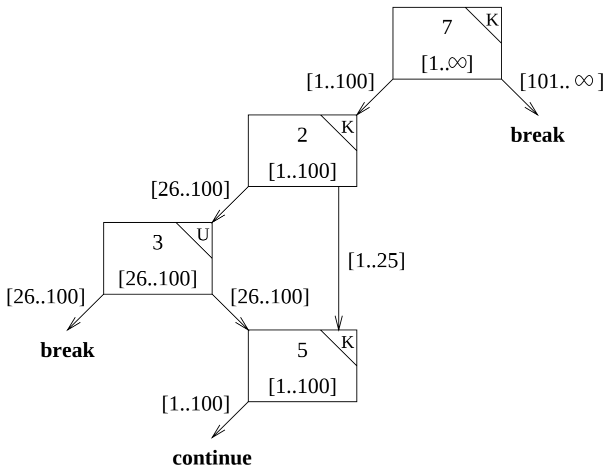


Figure 5. DAG of Branches with Ranges of Iterations

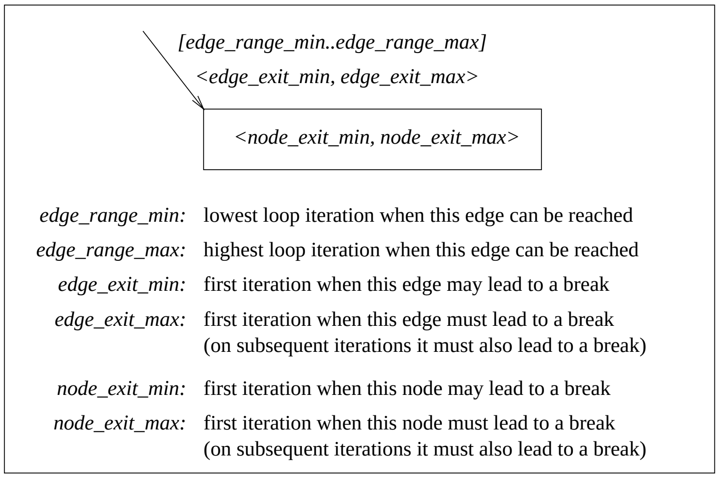
3.4. Determining the Minimum and Maximum Loop Iterations

The ranges of iterations associated with each node and edge of the DAG can be used to calculate the minimum and maximum number of iterations for the loop. To determine the minimum and maximum iteration value for each iteration branch, the DAG is processed in bottom-up order. The algorithm requires $O(I)$ complexity, where I is the number of iteration branches. The minimum and maximum iteration values for the root node of the DAG will be the minimum and maximum iteration values for the entire loop. Figure 6 defines the notation used in this subsection. Note that the range has been calculated using the technique presented in Section 3.3.

The following rules are used to assign minimum and maximum iteration values to edges.

1. If an edge is pointing to a *break*, then both the *edge_exit_min* and *edge_exit_max* are assigned the value of *edge_range_min*. (If there is a transition to a *break*, then the loop can only make that transition once.) This is the only point where a *bounded* value can be introduced since these are the only points where the loop can exit.

2. If an edge is pointing to a *continue*, then the *edge_exit_min* and *edge_exit_max* values for that edge are marked as *unbounded*, which


[edge_range_min..edge_range_max]
<edge_exit_min, edge_exit_max>
<node_exit_min, node_exit_max>

edge_range_min:	lowest loop iteration when this edge can be reached
edge_range_max:	highest loop iteration when this edge can be reached
edge_exit_min:	first iteration when this edge may lead to a break
edge_exit_max:	first iteration when this edge must lead to a break (on subsequent iterations it must also lead to a break)
node_exit_min:	first iteration when this node may lead to a break
node_exit_max:	first iteration when this node must lead to a break (on subsequent iterations it must also lead to a break)

Figure 6. Notation Used in Rules for Assigning Iteration Values

we will represent with '_'. (These transitions do not supply any information about when the loop exits.)

3. If the iteration branch associated with a node is classified as *known*, then the *node_exit_max* for the node is set to the smallest of the *bounded edge_exit_max* values on the outgoing edges or is denoted as *unbounded* if both outgoing edges have *unbounded edge_exit_max* values. (The loop has to exit when it will encounter a *break*.)

4. If the iteration branch associated with a node is classified as *unknown*, then the *node_exit_max* for the node is set to the largest of the *edge_exit_max* values on the outgoing edges of the node or is denoted as *unbounded* if either outgoing edge has an *unbounded edge_exit_max* value. (Use the largest value when it is not guaranteed that the node will actually reach the exit associated with a lower value.)

5. The *node_exit_min* for a node is set to the smallest of the *bounded edge_exit_min* values on the outgoing edges of the node or is denoted as *unbounded* if both outgoing edges have *unbounded edge_exit_min* values. (The smallest value represents the first possibility to exit the loop.)

6. An edge not leading to a *break* or *continue* is an edge leading to a node representing an iteration branch. The *edge_exit_min* and *edge_exit_max* values assigned to the edge depend upon one of three possible relations between the range of the edge and the iteration values of the node. These relations and the corresponding

edge assignments are depicted in Table IV. For example, the edge assignment when *node_exit_min* satisfies case 1 and *node_exit_max* satisfies case 2 would be ⟨*edge_range_min*, *node_exit_max*⟩. Case 1 depicts that the *edge_exit* is set to *edge_range_min* since this is the first iteration the edge can be traversed when the edge may lead to a *break*. Case 2 shows that the *edge_exit* is set to the *node_exit* when it is within the range of iterations that the edge is executed. Case 3 illustrates that the *edge_exit* is set to *unbounded* when there is no iteration on which the edge will be traversed after the edge can lead to a *break*.

Table IV. Rules for Assigning Iteration Values to an Incoming Edge

Case	Condition	Test	Edge_Exit
1	● ———	node_exit < edge_range_min	edge_range_min
2	——●——	edge_range_min ≤ node_exit & node_exit ≤ edge_range_max	node_exit
3	——— ●	edge_range_max < node_exit	_

Legend: ——— = [edge_range_min .. edge_range_max]
● = node_exit (i.e. node_exit_min or node_exit_max)

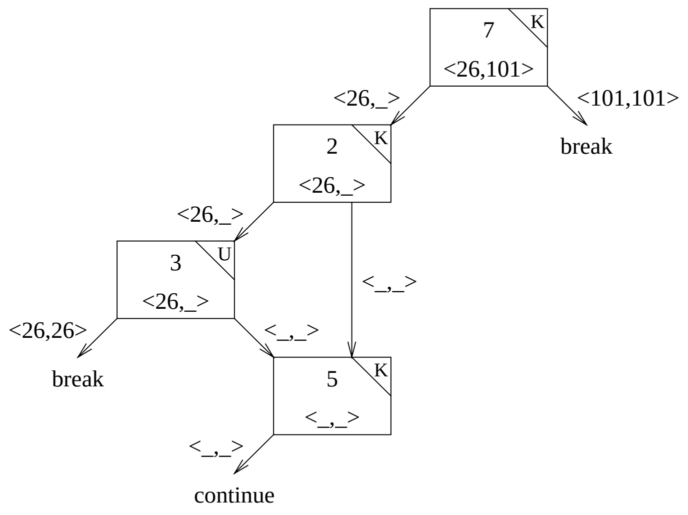


Figure 7. DAG of Iteration Branches with Minimum and Maximum Iteration Values

Figure 7 shows the same DAG as in Figure 5 on page 11, but with minimum and maximum iteration values assigned to edges and nodes. Node 5 and its incoming edges are assigned *unbounded* values since

```
for (i = 0; i != 100; i++)
   A;
```

(a) Bounded Loop

```
for (i = 0; ; i++) {
   if (i < 100 && somecond)
      continue;
   if (i == 50)
      break;
   }
```

(b) Potentially Unbounded Loop

```
for (i = 0; i != 100; i += 3)
   A;
```

(c) Unbounded Loop

Figure 8. Examples of Loops with Iteration Branches Using Equality Operators

there is no transition to a *break* for the range of loop iterations in which they are executed. Node 3 is assigned a minimum iteration value of 26 since that is the first possible iteration at which the node can take a transition to a *break*. Node 3's maximum iteration value is *unbounded* since node 3's iteration branch is classified as *unknown* and there is no guarantee that the transition to the *break* from node 3 will ever be taken. The minimum and maximum iterations for the entire loop is 26 and 101, respectively, since these are the iteration values in node 7, which is the root exit condition.

3.5. Supporting Iteration Branches Using Equality Operators

As stated in Table I on page 8, an iteration branch using an equality operator (i.e. `==` or `!=`) was earlier described as always being treated as an *unknown* branch. This may result in looser, but safe iteration bounds for loops containing equality operators. One reason for not addressing iteration branches that use the equality operators is that they may cause loop iteration ranges to become noncontiguous and would complicate the algorithms for bounding the number of iterations. However, in many cases iteration branches with equality operators can be handled using only contiguous ranges of iterations. For instance, Figure 8(a) contains a loop with an equality operator that our implementation was able to successfully bound. Our implementation classifies iteration branches with equality operators as *known* when the following three additional requirements to those specified in Table I are satisfied:

1. First, every path ending in a back edge in the loop must include the iteration branch with the equality operator. Figure 8(b) shows

Table V. How to Determine When an Equality Test Changes Direction

Operator	Condition	Test Result	*adjust*
==	$first < limit$ & $incr > 0$	is true on the Nth iteration	-1
==	$first > limit$ & $incr < 0$	is true on the Nth iteration	1
==	$first = limit$ & $incr = 0$	always true	
==	$first = limit$ & $incr \neq 0$	is false on the 2nd iteration	
==	otherwise	always false	
!=	$first < limit$ & $incr > 0$	is false on the Nth iteration	-1
!=	$first > limit$ & $incr < 0$	is false on the Nth iteration	1
!=	$first = limit$ & $incr = 0$	always false	
!=	$first = limit$ & $incr \neq 0$	is true on the 2nd iteration	
!=	otherwise	always true	

Where $first = initial + before$, $incr = before + after$,
N is defined in Equation 1, and *adjust* is used in Equation 1.

an example of a loop that may not execute the test for equality on the iteration in which the loop could exit.

2. Next, one of the outgoing transitions of the iteration branch with an equality operator must be to a *break*.

3. Finally, the following expression, which is part of Equation 1, must result in an integral value.

$$\frac{limit_i - (initial_i + before_i)}{before_i + after_i}$$

In other words, the *variable* must equal the *limit* of the iteration branch on some iteration. Figure 8(c) depicts a situation where the *variable* `i` will be assigned values (0, 3, ... 99, 102, ...) that will skip over the *limit* (100).

When the above requirements are fulfilled we have to check the initial value and increments to the variable, similar to Table III on page 9, and also choose a value for *adjust.* Table V shows when we can use Equation 1 on page 7 to determine on which iteration an equality test will change direction. Note that when $before+after = 0$ we need not use Equation 1 and thus we do not cause a divide by zero exception.

4. Supporting a Non-constant Loop-Invariant Number of Iterations

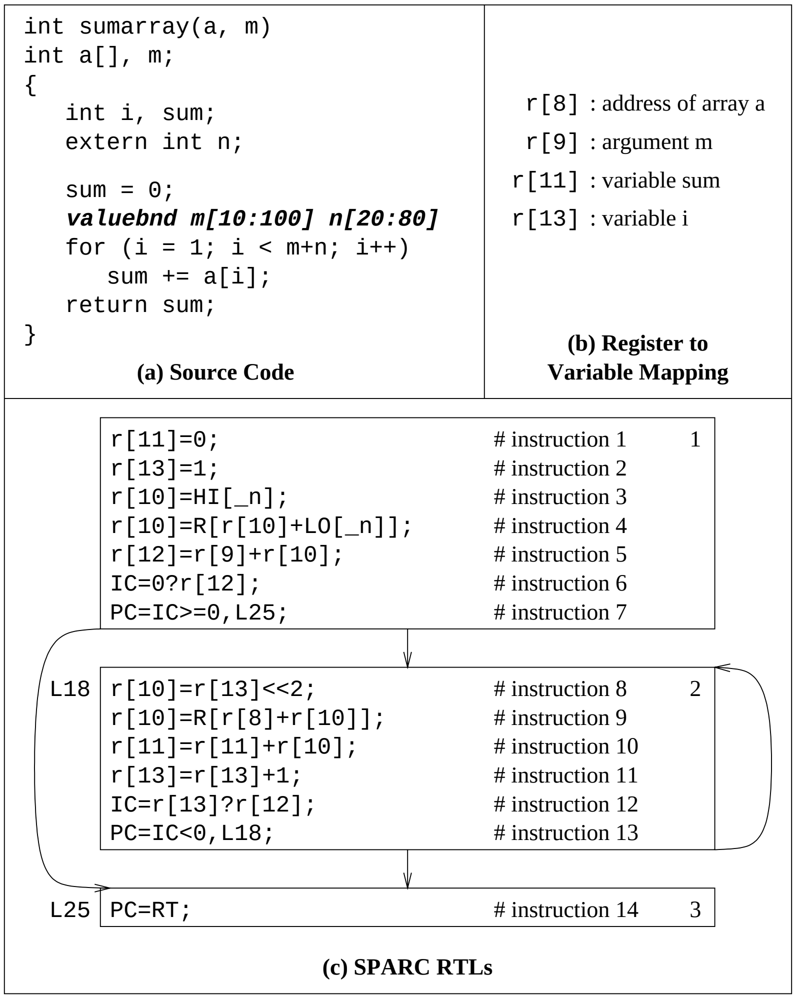
Sometimes a bounded number of iterations for a loop cannot be determined since the loop exit conditions involve the values of variables. Traditionally, timing analyzers have resolved this problem by requiring a user to specify the maximum number of iterations for a loop interactively (Park and Shaw, 1991; Li et al., 1995) or as an assertion in the source code. (Burns et al., 1996; Puschner and Koza, 1989) Unfortunately, there is no guarantee that the user will specify the correct number of iterations. Compilers may employ different code generation strategies or compiler optimizations that can affect the number of loop iterations. Thus, even an astute user may specify the number of loop iterations incorrectly.

Frequently the variables on which the number of loop iterations depend are loop invariant. In this case, a loop-invariant expression is calculated to represent the number of loop iterations. Essentially, we will still use Equation 1 on page 7, but relax the requirement that the *limit* and *initial* values have to be constants. Figure 9 shows an example function and it's corresponding SPARC RTLs. (Some compiler optimizations, such as loop strength reduction, have not yet been performed to simplify the example.) In this example, the control variable for the loop is `r[13]` and the limit is `r[12]`, which is loop invariant. The block preceding the loop is examined to determine the expression associated with the limit, which is expanded in the following steps:

```
1. r[12]                      # from instruction 12
2. r[9]+r[10]                 # from instruction 5
3. r[9]+R[r[10]+LO[_n]]       # from instruction 4
4. r[9]+R[HI[_n]+LO[_n]]      # from instruction 3
5. m+n
```

The register `r[9]` has been allocated to the argument `m`, whose value was also passed to the function in the same register. The compiler remembers the register and the blocks where each live range of a local variable or argument is allocated to a register. Thus, the compiler was able to associate the register `r[9]` with the argument `m` and that the memory reference is to the global variable `n`. We use Equation 1 to generate a symbolic expression (containing the local variable `m` and global variable `n`) to represent the number of iterations, as shown in Figure 10.

When the compiler can determine that the number of iterations is non-constant and loop invariant, the loop-invariant expression is passed



```
int sumarray(a, m)
int a[], m;
{
   int i, sum;
   extern int n;

   sum = 0;
   valuebnd m[10:100] n[20:80]
   for (i = 1; i < m+n; i++)
      sum += a[i];
   return sum;
}
```

(a) Source Code

```
 r[8] : address of array a
 r[9] : argument m
r[11] : variable sum
r[13] : variable i
```

(b) Register to Variable Mapping

```
    r[11]=0;                    # instruction 1     1
    r[13]=1;                    # instruction 2
    r[10]=HI[_n];               # instruction 3
    r[10]=R[r[10]+LO[_n]];      # instruction 4
    r[12]=r[9]+r[10];           # instruction 5
    IC=0?r[12];                 # instruction 6
    PC=IC>=0,L25;               # instruction 7

L18 r[10]=r[13]<<2;             # instruction 8     2
    r[10]=R[r[8]+r[10]];        # instruction 9
    r[11]=r[11]+r[10];          # instruction 10
    r[13]=r[13]+1;              # instruction 11
    IC=r[13]?r[12];             # instruction 12
    PC=IC<0,L18;                # instruction 13

L25 PC=RT;                      # instruction 14    3
```

(c) SPARC RTLs

Figure 9. Loop with a Non-constant Loop-Invariant Number of Iterations

to the timing analyzer. The user is prompted by the timing analyzer for the minimum and maximum values for each variable in this expression. To simplify identification of these variables, the timing analyzer also informs the user of the function and line number associated with the loop. After receiving the minimum and maximum values for these variables, the timing analyzer automatically calculates the minimum and maximum number of loop iterations.[3]

[3] Note that the timing analyzer will not permit the number of iterations to be fewer than 1. In the above example, a user may indicate that the minimum values of `m` and `n` are both 0. Simply substituting these values in the expression would result in the number of loop iterations being -1. But if the loop is entered, then it has to

$$\begin{aligned} N &= \left\lfloor \frac{\mathit{limit} - (\mathit{initial} + \mathit{before}) + \mathit{adjust}}{\mathit{before} + \mathit{after}} \right\rfloor + 2 \\ &= \left\lfloor \frac{\mathtt{m} + \mathtt{n} - (1+1) + -1}{1+0} \right\rfloor + 2 \\ &= \mathtt{m} + \mathtt{n} - 1 \end{aligned}$$

Figure 10. Finding a Symbolic Bound for Example in Figure 9

The authors also modified the compiler to allow the user to specify assertions about the minimum and maximum values of variables associated with loops. The boldface line in Figure 9(a) contains assertions for the minimum and maximum values of the variables `m` and `n`. The compiler uses the loop-invariant expression and replaces the variables with the minimum and maximum specified values. The minimum number of iterations of 29 and the maximum number of iterations of 179 is automatically passed to the timing analyzer and no user intervention is required. Of course, the analysis will only be as accurate as the assertions themselves.

When a loop-invariant expression cannot be calculated, the timing analyzer will prompt the user for the minimum and maximum number of iterations instead of values of variables. However, we have found that a constant or loop-invariant number of iterations can be typically calculated for most loops in the numerical benchmarks and applications we have examined.

5. Bounding Iterations for Non-Rectangular Loop Nests

The previous sections described approaches to determine the minimum and maximum number of iterations for a loop, given that the number of iterations depends only upon either constant or loop-invariant values. Unfortunately, many nested loops do not fulfill this requirement.

In this section we will describe a novel method to determine the number of iterations for a nested loop whose iteration bound depends upon the loop index of an outer loop. Such a loop nest is called *non-rectangular*. A typical example of a non-rectangular loop nest is the loop nest of the bubble sort program in Figure 11.

execute at least one iteration since the number of iterations is defined as the number of times the loop header block is executed.

```
for(i = 0; i < 99; i++)
   for(j = i+1; j < 100; j++)
      if(a[i] > a[j]) swap(a,i,j);
```

Figure 11. A Typical Non-Rectangular Loop Nest

Non-rectangular loop nests have long presented a problem for timing analyzers since the resulting timing predictions are typically quite loose (Healy et al., 1995; Hur et al., 1995; Li et al., 1995). In fact, these overly pessimistic predictions may indicate that a program does not meet its timing constraints, when it actually does.

This section describes a general and efficient method for obtaining tight timing predictions for non-rectangular loop nests usually encountered in programs. This is accomplished by formulating the number of loop iterations in terms of summations, where each summation represents the number of iterations to be executed by a loop. Such an equation can be efficiently solved given that certain restrictions are met.

5.1. Formulating the Number of Iterations

In this subsection we describe how a loop nest may be formulated in terms of summations. The framework we present was based on work by Sakellariou (Sakellariou, 1997; Sakellariou, 1996). The number of iterations of a single loop, where the loop variable is incremented by one (so called *unit stride*), can be represented by a summation when the lower bound (a) is less than or equal to the upper bound (b), as shown in Equation 2.

$$N = \sum_{i=a}^{b} 1 = \begin{cases} b - a + 1 & \text{if } a \leq b \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Figure 12 shows how two different loop nests can be formulated in terms of summations. The total number of iterations to be executed by the innermost loop in each loop nest are calculated by solving the corresponding equation. The Bernoulli formula shown in Equation 3, where $p \geq 1$ and $n \geq 1$ and B_k is the Bernoulli number of order k, can be used to evaluate terms in a summation.

$$\sum_{i=1}^{n} i^p = \frac{1}{p+1} \sum_{k=0}^{p} \binom{p+1}{k} B_k (n+1)^{p-k+1} \tag{3}$$

```
for (i=1; i<99; i++)
    for (j=i+1; j<100; j++)
        A;
```

$$\begin{aligned}
N &= \sum_{i=1}^{98} \sum_{j=i+1}^{99} 1 \\
&= \sum_{i=1}^{98} \left(\sum_{j=1}^{99} 1 - \sum_{j=1}^{i} 1 \right) \\
&= \sum_{i=1}^{98} (99 - i) \\
&= \sum_{i=1}^{98} 99 - \sum_{i=1}^{99} i \\
&= 4851
\end{aligned}$$

(a) Loop Nest from Sort Program

```
for (j=1; j<=100; j++)
    for (i=j; i<=100; i++)
        for (k=1; k<j; k++)
            A;
```

$$\begin{aligned}
N &= \sum_{j=1}^{100} \sum_{i=j}^{100} \sum_{k=1}^{j-1} 1 \\
&= \sum_{j=1}^{100} \sum_{i=j}^{100} (j-1) \\
&= \sum_{j=1}^{100} \left(\sum_{i=1}^{100} (j-1) - \sum_{i=1}^{j-1} (j-1) \right) \\
&= \sum_{j=1}^{100} \left(\sum_{i=1}^{100} j - \sum_{i=1}^{100} 1 \sum_{i=1}^{j-1} j + \sum_{i=1}^{j-1} 1 \right) \\
&= \sum_{j=1}^{100} \left(102j - j^2 - 101 \right) \\
&= 102 \sum_{j=1}^{100} j - \sum_{j=1}^{100} j^2 - 101 \sum_{j=1}^{100} 1 \\
&= 166650
\end{aligned}$$

(b) Loop Nest from LU Decomposition Program

Figure 12. Deriving the Total Number of Iterations for Two Loop Nests

The constraint on the bounds in Equation 2 results from the fact that the value of the sum must equal 0 if the lower bound a is greater than the upper bound b. The explicit constraint is necessary to accurately count the number of iterations of so-called *zero-trip* loops. Zero-trip loops do not execute the loop body when the lower bound exceeds the upper bound, given that the stride is positive.

We can represent summations with non-unit strides, where the stride s is specified along with the lower bound a and upper bound b. Equation 4 shows how a non-unit stride can be used in a conventional summation, where E is an expression and $E[i \leftarrow si + a]$ denotes the substitution of all free occurrences of i by $si+a$. This is effectively a change in variables and does not change the value of the summation. The change allows summations with strides to be represented by normalized summations

(summations with stride 1).

$$I = \sum_{i=a}^{b,s} E = \sum_{i=0}^{\lfloor (b-a)/s \rfloor} E[i \leftarrow si + a] \tag{4}$$

Summations with non-unit strides are more difficult to evaluate since one has to deal with summations of floors. Equation 5 shows how a floor can be converted to an expression involving a modulo operation (%). A modulo operation can often be simplified using Equation 6 (Sakellariou, 1996).

$$\left\lfloor \frac{n}{m} \right\rfloor = \frac{n - n\%m}{m} \text{ , if } m > 0 \text{ \& } n > 0 \tag{5}$$

$$\sum_{i=0}^{n} (i\%d)^p = \begin{cases} \sum_{i=0}^{n} i^p & \text{if } n < d \\ \sum_{j=0}^{\lfloor n/d \rfloor -1} \sum_{i=0}^{d-1} + i^p \sum_{i=0}^{n\%d} i^p & \text{if } n \geq d \end{cases} \tag{6}$$

However, summations involving modulo operations are more difficult to simplify when two or more loops have non-unit strides and the bounds are symbolic. Fortunately, this situation rarely occurs. Equations 2–6 can be used to correctly determine that the total iterations for the loop nest in Figure 13 is 1717. Unfortunately, sometimes an expression in a summation may contain a product of two or more terms containing modulo operations. In this case, an approximation of the iteration count is used, which is shown in Equation 7.

$$\sum_{i=a}^{b,s} E \approx \sum_{i=a}^{\lfloor b/s \rfloor} E/s \tag{7}$$

```
for (i=0; i<100; i++)
    for (j=i; j<100; j+=3)
        A;
```

Figure 13. A Loop Nest Containing a Non-unit Stride

As suggested by Sakellariou (Sakellariou, 1996; Sakellariou, 1997), a computer algebra system can be exploited off line to solve the equations of summations. However, computer algebra systems, such as *Maple* (Char et al., 1988), give inaccurate results when the bounds restriction on the summation is violated in Equation 2. In general, every loop iteration count problem that is cast as a summation should evaluate to zero if the lower bound is greater than the upper bound. However, it is not always possible to evaluate the test when the bounds are symbolic. For example, consider the loop nest in Figure 14. The inner loop is a zero-trip loop for values of i greater than 2. We define a *partially zero-trip* loop to be a loop that is zero-trip depending on values of index variables of outer loop(s). By applying Equation 2, the iteration count of the partially zero-trip loop can be defined as shown in Figure 14. Clearly, the result is $N = 3$. However, a naive evaluation without the bounds test results in $N = -7$. This means that when a computer algebra system is to be used off line, the summations should be guarded with bounds tests. Unfortunately, computer algebra systems cannot effectively deal with the simplification of nested summations with additional tests on the bounds of inner summations. The reason is that the test may be symbolic, as shown in Figure 14. The solution is to isolate possible conditions on the iteration variable from the test and to simplify summations as shown in Equation 8 for any expression e. Note that c may not necessarily lie within the range $[a..b]$ and relations besides $<$ may be used.

$$\sum_{i=a}^{b} \begin{cases} E & \text{if } i < c \\ 0 & \text{otherwise} \end{cases} = \begin{cases} \sum_{i=a}^{\min(b,c)} E & \text{if } a < c \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

```
for (i=1; i<8; i++)
    for (j=i; j<3; j++)
        A;
```

$$N = \sum_{i=1}^{7} \begin{cases} 3 - i & \text{if } i < 3 \\ 0 & \text{otherwise} \end{cases}$$

Figure 14. A Partially Zero-Trip Loop

5.2. Implementation

The implementation for evaluating the summations described in the previous section was accomplished by using the algebraic simplifier portion of the Ctadel system (van Engelen et al., 1996; van Engelen et al., 1997). The authors' timing analyzer (Healy et al., 1999) and Ctadel were compiled separately, but Ctadel is directly integrated into the timing analyzer by linking the object files. This avoids unnecessary overhead that would result from passing expressions between the timing analyzer and Ctadel by operating systems calls. The summations are formulated in the timing analyzer and Ctadel is invoked as a C function with the summation parameters as arguments.[4]

Another complication when dealing with zero-trip loops in the timing analyzer is due to the way the timing analyzer counts iterations. As mentioned in Section 3.1, the number of loop iterations is the number of times the loop header is executed, as opposed to the number of times the loop body is encountered. Thus, when a loop is entered, it is guaranteed to iterate at least once. The zero-trip case in Equation 8 can be modified to indicate a single iteration, as shown in Equation 9.

$$\sum_{i=a}^{b} \begin{cases} E & \text{if } i < c \\ 1 & \text{otherwise} \end{cases} = \\ \begin{cases} \sum_{i=a}^{\min(b,c-1)} E & \text{if } a < c \\ 0 & \text{otherwise} \end{cases} + \begin{cases} \sum_{i=\max(a,c)}^{b} 1 & \text{if } c \leq b \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

Figure 15 shows how the loop nest in Figure 14 can be formulated as a summation and solved to produce an accurate number of iterations. Note that the test in Figure 15 has iteration variable i isolated to the left of the relation. An isolation algorithm is used by Ctadel to analyze the test and isolate the variable.

It is known that the detection of zero-trip loops in the general case is NP-complete, because it amounts to solving a integer linear programming problem. Similarly, adjusting the bounds of loops to avoid partially zero-trip loops is NP-complete. This normalization process can be performed with the Fourier-Motzkin (FM) elimination method (Wolfe, 1996). However, one can argue that real-world algorithms rarely exhibit (partially) zero-trip loops, because algorithms with partially zero-trip loops are deemed to be inefficient.

The timing analyzer verifies that there are no zero-trip loops for an inner loop by expanding its initial value and limit. Likewise, the timing

[4] The authors have created a Web page demonstrating the functionality of the Ctadel. It can calculate the number of loop iterations for a loop nest specified by the user. The URL is `http://www.cs.fsu.edu/~engelen/iternum.cgi`.

$$\begin{aligned}
N &= \sum_{i=1}^{7} \begin{cases} 3-i & \text{if } i<3 \\ 1 & \text{otherwise} \end{cases} \\
&= \sum_{i=1}^{2}(3-i) + \sum_{i=3}^{7} 1 \\
&= 3+5 \\
&= 8
\end{aligned}$$

Figure 15. Deriving the Number of Iterations for the Loop Nest in Figure 14

analyzer is able to verify that there are no partially zero-trip loops in the loop nest. However, if the verification is inconclusive, the loop nest may or may not contain (partial) zero-trip loops. For instance, consider the loop nest in Figure 16. The expansion of the innermost loop initial value and limit is depicted in Figure 17. The timing analyzer is able to guarantee that the inner loop is not zero-trip since the initial value is never greater than the limit.

```
for (i=0; i<10; i++)
    for (j=i; j<11; j++)
        for (k=i-3; k<j+8; k++)
            A;
```

Figure 16. Innermost Loop Detected Zero-Trip Free by the Timing Analyzer

Initial Value	Limit
$i-3$	$j+8$
$[0..9]-3$	$[i..10]+8$
$[-3..6]$	$[[0..9]..10]+8$
	$[0..10]+8$
	$[8..18]$

Figure 17. Expanding Initial and Limit Values of Innermost Loop in Figure 16

Now consider the loop in Equation 18 and the corresponding expansion of the initial value and limit in Figure 19. The test is inconclusive. However, the loop nest is not zero-trip due to the $j < i$ condition in the middle loop. Since the range analysis can be used to safely verify if a loop is partially zero-trip, it is possible to use the results in deciding which summation solver to use. For example, the loop in Figure 16 can be safely cast into a summation without a bounds tests, while the summations for the loop in Figure 18 requires a bounds test (see Figure 15 for an example bounds test). The disadvantage of having a bounds test is that a loop with a stride poses problems for solving the summation because the summation bounds test may contain modulo operations on the iteration variable, which prohibits the application of Equation 9.

```
for (i=1; i<10; i++)
    for (j=0; j<i; j++)
        for (k=j; k<i; k++)
            A;
```

Figure 18. Innermost Loop Nest Detected Zero-Trip Free by CTADEL

Initial Value	Limit
j	$i - 1$
$[0..i]$	$[1..9] - 1$
$[0..[1..9]]$	$[0..8]$
$[0..9]$	

Figure 19. Expanding Initial and Limit Values of Innermost Loop in Equation 18

The timing analyzer decides among three possible solution methods to evaluate the summation representing a loop nest:

– CTADEL evaluates the summation while testing the bounds of the index variables.

– CTADEL evaluates the summation without testing for bounds.

– The timing analyzer derives conservative lower and upper bounds on the sum, based on constant bounds given in outer level loops.

The algorithm for selecting the appropriate method is described in Figure 20. The exact solutions are computed using safe assumptions in the possible presence of partially zero-trip loops, using either method (1) or (2). This algorithm will resort to method (3) only in the presence of multiple loops with non-unit strides.

```
The timing analyzer verifies that the loop nest is not (partially) zero-trip.
IF the check is successful THEN
    The loop nest is formulated into summation without bounds tests and
    presented to CTADEL.
    IF CTADEL is able to solve the summation THEN
        RETURN the integer count.
    ELSE
        CTADEL could not solve the summation in the presence of two or more
        loops with non-unit strides.
        RETURN conservative lower and upper bounds on the sum.
    END IF
ELSE
    The check is inconclusive and the loop nest is cast into a summation with
    bounds tests.
    The rewritten summation is presented to CTADEL.
    IF CTADEL is able to solve the summation THEN
        RETURN the integer count.
    ELSE
        CTADEL could not solve the summation in the presence of two or more
        loops with non-unit strides.
        RETURN conservative lower and upper bounds on the sum.
    END IF
END IF
```

Figure 20. Algorithm for Selecting a Solution Method for Summations

The following approach is used in the timing analyzer to obtain tight predictions of non-rectangular loop nests whose total iterations in a loop nest are known. The timing analyzer calculates WCET and BCET predictions based on the maximum and minimum number of iterations, respectively, for the loop whose number of iterations varies. These predictions are made in case a user requests the WCET or BCET predictions for the loop. In addition to these absolute predictions, the timing analyzer also calculates *average* WCET and BCET predictions for each loop. To calculate the average number of iterations for a loop, we divide the total iterations by the total number of times the loop is entered. For instance, in the previous subsection we found that the total number of iterations for the innermost loop from the *sort* program in

Figure 12 on page 20 was 4851. We also calculate the number of times the current loop is entered by calculating the total number of iterations for the loop that encloses the current loop. In this example, the innermost loop is entered 98 times. Thus, the average number of iterations for the loop is 49.5 (4851/98). The average number of iterations is used to calculate the average WCET and BCET predictions. When a non-integer is calculated, we round up for the WCET prediction and truncate for the BCET prediction since our loop analysis algorithm is designed to work on an integral number of iterations.

5.3. Results

Table VI shows programs that were evaluated using the approach of calculating an average number of iterations for loops. These programs benefit from using this approach since they each contain one or more non-rectangular loop nests. Note that the *Sort* program has been used in the past as one of the test programs to evaluate our timing analyzer (Arnold et al., 1994; Healy et al., 1995; White et al., 1997). The size of a program is measured as number of assembly instructions in the compiled and optimized program.

Table VI. Test Programs Containing Non-Rectangular Loop Nests

Name	Description or Emphasis	Size
Hes	Reduces a 100x100 matrix to Hessenberg Form	221
Integ	Evaluates a Double Integral over a Trapezoidal Region	45
Interp	Polynomial Interpolation of 500 Points	178
LU	LU Decomposition of a 100x100 Matrix	278
Sort	Bubble sort of 500 Integers	130
Sym	Tests If a 500x500 Matrix Is Symmetric	50

Table VII shows the best and worst-case cycles required for executing with instruction caching and pipelining for the MicroSPARC I (Texas Instruments, 1993). The *previous ratio* and *current ratio* columns show that when the timing analyzer used the average inner loop predictions, the predicted execution times were significantly tighter. *Interp* showed a significant improvement in best case since the best case number of iterations for the inner loop of a non-rectangular loop nest was 1, which was significantly lower than the average number of iterations. If the timing analyzer did not use an average number of inner loop iterations in worst case, then the number of loop iterations for the triangular loops in *Interp*, *Sort*, and *Sym* would have been approximately double. The WCET of these programs are nearly exact using the average number of iterations. The *Integ* program had a higher best-case *previous ratio* and a lower worst-case *previous ratio* since there were other loops in this program that contributed more significantly to the total execution time. The *Sort* and *Sym* programs did not have a significant underestimation (i.e. *previous ratio*) in best case. In the best case for *Sort* the values were initially sorted and the sort function exited once the array has been detected to be in ascending order. Likewise, the *Sym* program terminates when it finds the first pair of values that are not equal. *Hes* and *LU* are unlike the other programs in that they contain some triply nested loops. In some loop nests the loop variables of the innermost and middle loops depend on the outermost index variable. In other loop nests the innermost loop variable depends on the loop variable of the middle loop, which in turn depends on the loop variable of the outer loop. CTADEL correctly determines the exact number of loop iterations in all of these cases and the results are more accurate WCET predictions compared to its *previous ratio*s. However, the improvement in BCET for *LU* was less substantial.[5]

Table VIII shows the response time of the timing analyzer for each of the test programs. To obtain these measurements, the timing analyzer was invoked for each test program ten times on a Sun HPC 3000 processor. The figures in the table represent the averages of the ten trials. Note that the times reported here include the analysis of both best and worst case predictions, which occurred in the same invocation of the analyzer. We found that the number of conditional constructs (e.g. *if* statements) rather than the number of loops and functions, tends to have the biggest impact on the analysis time since it affects the number of paths that must be analyzed.

6. Coding Conventions to Make Loop Bounds Predictable

We have found that a programmer can write code where the timing analyzer can accurately determine the number of iterations when the following conventions are used. We do realize that these conventions

[5] The timing predictions for the *Hes* and *LU* programs are still fairly loose. This is primarily due to the fact that several loops were preceded by guards resulting from `if` statements and loop code generation strategies. Each of these guards tests the value of a loop control variable. The authors have recently done work in detecting this type of constraint (Healy and Whalley, 1999b) when dealing with rectangular loop nests. We anticipate to extend this analysis to non-rectangular loop nests for the final version of this paper. This ability to detect constraints on loop control variables should substantially tighten both the WCET and BCET predictions for the *Hes* and *LU* programs.

Table VII. Timing Analysis Results

Best-Case Results

Name	Observed Cycles	Previous Estimated Cycles	Previous Ratio	Current Estimated Cycles	Current Ratio
Hes	306,341	13,614	0.044	256,516	0.837
Integ	19,160,842	12,785,618	0.667	19,135,118	0.999
Interp	6,485,878	143,064	0.022	6,479,865	0.999
LU	13,792,698	278,683	0.020	637,383	0.046
Sort	19,966	19,950	0.999	19,950	0.999
Sym	160	160	1.000	160	1.000

Worst-Case Results

Name	Observed Cycles	Previous Estimated Cycles	Previous Ratio	Current Estimated Cycles	Current Ratio
Hes	55,747,317	130,932,770	2.281	57,389,258	1.029
Integ	22,538,082	30,023,163	1.332	22,553,163	1.001
Interp	25,469,403	50,702,358	1.991	25,479,405	1.000
LU	22,436,763	141,900,455	6.324	26,410,255	1.177
Sort	7,672,281	15,251,603	1.988	7,672,292	1.000
Sym	2,747,654	5,481,220	1.995	2,747,698	1.000

cannot always be used for some programs, such as non-numerical applications. However, we believe these conventions can be followed for most numerical applications.

1. When possible, make loop exit conditions only dependent on loop counter variables.

2. Use local integer variables for loop counter variables.

3. Try not to increment or decrement loop counter variables in conditionally executed code.

4. When possible, use integer constants for the initial value, limit, and increments of loop counter variables. Otherwise, try to use loop invariant values.

5. Try to avoid non-unit strides in non-rectangular loop nests.

Table VIII. Analysis Response Times in Seconds

Name	Analysis Time
Hes	0.73
Integ	0.14
Interp	0.36
LU	1.10
Sort	0.25
Sym	0.22
Average	0.47

6. Try to avoid conditionally executed loops in non-rectangular loop nests.

7. Conclusions

In this paper we have presented three different methods for bounding the number of iterations of a loop. First, a method was described that determines the minimum and maximum number of iterations of loops with multiple exits and also detects infeasible paths. For instance, loops of the form in Figure 21(a) that can exit prematurely when some condition becomes true are quite common and the bounded number of iterations of such loops can be detected by the general algorithm presented in the paper.

Second, a method to derive a symbolic expression representing the number of iterations is presented. The symbolic expression is used to bound the number of iterations of loops which have a non-constant number of iterations. Figure 21(b) shows an example of this common type of loop. The user can specify the minimum and maximum values of the variables in the symbolic expression by placing assertions in the source code or by interactively responding to prompts from the timing analyzer. These assertions are more reliable than specifying the minimum and maximum number of loop iterations directly since the user does not have to be aware of the code generation strategies or optimizations performed by the compiler. Also, if value range analysis of variables is deployed the bounds of the variables can be automatically provided by the compiler.

```
for (i = 0; i < 100; i++) {
   ...
   if (somecond)
      break;
   ...
   }
```

(a) Loop with Multiple Exits

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

(b) Loop with a Nonconstant Number of Iterations

```
for (i = 0; i < 99; i++)
   for (j = i+1; j < 100; j++) {
      ...
      }
```

(c) Inner Loop Whose Number of Iterations Depends on an Outer Loop Counter Variable

Figure 21. Common Forms of Loops

Finally, timing analysis support is given to tightly predict the execution time of a non-rectangular loop nest, i.e. a loop nest where the number of iterations of an inner loop is dependent on counter variables of outer level loops. These loop nests, such as the one shown in Figure 21(c), appear frequently in programs and can result in significant overestimations in worst-case predictions (as well as underestimations in best-case predictions). Our approach more tightly predicts the number of iterations when the initial value or limit of the control variable in an inner loop depends on a control variable of an enclosing outer loop.

```
IF A loop variable has a non-constant loop-invariant initial value, limit, or stride
   that is not dependent on an outer loop variable AND
   There are no other loop variables to bound the number of loop iterations THEN
      Use information provided by the user (assertions or responses to queries) as
      described in Section 4 to obtain bounds on these variables.
END IF
Calculate the minimum and maximum iterations as described in Section 3.
IF The value of the loop variable is dependent on an outer loop variable THEN
      Calculate an average number of iterations for the loop,
      using the techniques described in Section 5.
END IF
```

Figure 22. Algorithm for Selecting a Solution Method for Bounding Loop Iterations

Figure 22 shows the algorithm used to decide which of the techniques presented in this paper use for a particular loop.

These methods have been successfully integrated in an existing compiler and an associated timing analyzer that predicts the performance for optimized code on a machine that exploits caching and pipelining. The result is tighter and more reliable timing analysis predictions and less work for the user.

Acknowledgements

The authors thank Jack Davidson for allowing *vpo* to be used for this research. Manuel Benitez implemented the original algorithm in *vpo* to calculate the number of iterations of a loop with a single exit condition. Frank Mueller provided several helpful suggestions on an earlier draft of this paper. Rizos Sakellariou provided several suggestions on a later draft. The conversations with Rizos Sakellariou about his work on calculating the total number of loop iterations for a non-rectangular loop nest proved to be invaluable. The authors also appreciate the suggestions from the anonymous reviewers, which measurably improved the quality of the paper. This work was supported in part by NSF grant EIA-9806525.

References

Aho, A. V., R. Sethi, and J. D. Ullman: 1986, *Compilers Principles, Techniques, and Tools*. Addison-Wesley.

Arnold, R., F. Mueller, D. Whalley, and M. Harmon: 1994, 'Bounding Worst-Case Instruction Cache Performance'. In: *Proceedings of the Fifteenth IEEE Real-Time Systems Symposium*. pp. 172–181.

Benitez, M. E. and J. W. Davidson: 1988, 'A Portable Global Optimizer and Linker'. In: *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*. pp. 329–338.

Burns, A., R. Chapman, and A. Wellings: 1996, 'Combining Static Worst-Case Timing Analysis and Program Proof'. *Real-Time Systems Journal* **11**, 145–171.

Char, B., K. Geddes, G. Gonnet, M. Monagan, and S. Watt: 1988, 'MAPLE Reference Manual'.

Ermedahl, A. and J. Gustafsson: 1997, 'Deriving Annotations for Tight Calculation of Execution Time'. In: *Proceedings of European Conference on Parallel Processing*. pp. 1298–1307.

Healy, C., R. Arnold, F. Mueller, D. Whalley, and M. Harmon: 1999, 'Bounding Pipeline and Instruction Cache Performance'. *IEEE Transactions on Computers* **48**(1), 53–70.

Healy, C. A. and D. B. Whalley: 1999a, 'Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints'. In: *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. pp. 79–88.

Healy, C. A. and D. B. Whalley: 1999b, 'Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints'. In: *Proceedings of the IEEE Real-Time Technology and Applications Symposium.*

Healy, C. A., D. B. Whalley, and M. G. Harmon: 1995, 'Integrating the Timing Analysis of Pipelining and Instruction Caching'. In: *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium.* pp. 288–297.

Hennessy, J. and D. Patterson: 1996, *Computer Architecture: A Quantitative Approach, Second Edition.* Morgan Kaufmann.

Hur, Y., Y. Bae, S. Lim, S. Kim, B. Rhee, S. Min, C. Park, M. Lee, H. Shin, and C. Kim: 1995, 'Worst Case Timing Analysis of RISC Processors: R3000/R3010 Case Study'. In: *Proceedings of the IEEE Real-Time Systems Symposium.*

Kligerman, E. and A. Stoyenko: 1986, 'Real-Time Euclid: A Language for Reliable Real-Time Systems'. *IEEE Transactions on Software Engineering* **12**(9), 941–949.

Lam, M.: 1988, 'Software Pipelining: An Effective Scheduling Technique for VLIW Machines'. In: *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation.* pp. 318–328.

Li, Y. S., S. Malik, and A. Wolfe: 1995, 'Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software'. In: *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium.* pp. 298–307.

Liu, Y. and G. Gomez: 1998, 'Automatic Accurate Time-Bound Analysis for High-Level Languages'. In: *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems.* pp. 31–40.

Lundqvist, T. and P. Stenström: 1998, 'Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques'. In: *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems.* pp. 1–15.

Park, C. Y. and A. C. Shaw: 1991, 'Experiments with a Program Timing Tool Based on a Source-Level Timing Schema'. *Computer* **24**(5), 48–57.

Puschner, P. and C. Koza: 1989, 'Calculating the Maximum Execution Time of Real-Time Programs'. *Real-Time Systems* **1**(2), 159–176.

Sakellariou, R.: 1996, 'On the Quest for Perfect Load Balance in Loop-Based Parallel Computations'. Ph.D. thesis, Department of Computer Science, University of Manchester.

Sakellariou, R.: 1997, 'Symbolic Evaluation of Sums for Parallelising Compilers'. In: *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics.* pp. 685–690.

Stone, H. S.: 1990, *High-Performance Computer Architecture, Second Edition.* Addison Wesley.

Texas Instruments, I.: 1993, 'Product Preview of the TMS390S10 Integrated SPARC Processor'.

van Engelen, R., L. Wolters, and G. Cats: 1996, 'Ctadel: A Generator of Multi-Platform High Performance Codes for PDE-based Scientific Applications'. In: *Proceedings of the 10th ACM International Conference on Supercomputing.* pp. 86–93.

van Engelen, R., L. Wolters, and G. Cats: 1997, 'Tomorrow's Weather Forecast: Automatic Code Generation for Atmospheric Modeling'. *IEEE Journal of Computational Science and Engineering* **4**(3), 22–31.

White, R. T., F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon: 1997, 'Timing Analysis for Data Caches and Set-Associative Caches'. In: *Proceedings of the IEEE Real-Time Technology and Applications Symposium.* pp. 192–202.

Wolfe, M. J.: 1996, *High Performance Compilers for Parallel Computers.* Addison-Wesley.