# Bounding Loop Iterations for Timing Analysis

Christopher Healy* Mikael Sjödin† Viresh Rustagi‡ David Whalley*

## Abstract

*Static timing analyzers 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. Second, the loop-invariant variables on which the number of loop iterations depends are identified for which the user can provide minimum and maximum values. Finally, a method is given to tightly predict the execution time of loops whose number of iterations is dependent on counter variables of outer level loops. These 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.*

## 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 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 [1]. Applications for determining this number include more efficient implementations of loop unrolling [2], software pipelining [3], and exploiting parallelism across loop iterations [4]. When the number of iterations cannot be exactly determined, it would be desirable in a real-time system to know the lower and upper iteration bounds. 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 [5, 6]. 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 [7, 8, 9]. 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 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 to vary by one iteration. Finally, compiler optimizations, such as loop unrolling, 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 desirable to have the compiler automatically and efficiently determine the bounds for each loop in a program when possible. Some work has been recently accomplished to determine the number of loop iterations automatically using abstract interpretation [10]. While this technique is quite powerful, it often results in significant analysis overhead.

This paper describes three approaches that support timing analysis by bounding the number of loop iterations. First, an algorithm is presented that determines a bounded number of iterations for loops with multiple exits. Second, the user can provide information for loop-invariant variables on which the number of loop iterations depends. Finally, a method is given to accurately predict the average number of iterations for loops whose number of iterations can vary depending upon the values of counter variables of enclosing outer loops. All three of these approaches are efficiently implemented and result in less work for a user. The last approach also results in tighter timing analysis predictions. These approaches were implemented by modifying the *vpo* compiler [1] to analyze loops and this loop analysis information is passed to a timing analyzer [11, 12, 13] to predict performance.

## 2. Iterations for Loops with Multiple Exits

In this section we present a method to determine a bounded number of iterations for natural loops with

* Computer Science Department, Florida State University, Tallahassee, FL 32306-4530, phone: (850) 644-3506, e-mail: {healy, whalley}@cs.fsu.edu

† Department of Computer Systems, Uppsala University, Sweden, phone: +46-18-4717605, e-mail: mic@docs.uu.se

‡ Objectime, Inc., 226 Airport Parkway, #480, San Jose, CA 95110, phone: (408) 441-1124, e-mail: vrustagi@objectime.com



Copyright 1998 IEEE. Published in the Proceedings of RTAS'98.

multiple exits. (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.

## 2.1. Identifying the Iteration Branches

Some terms are now defined to facilitate the presentation of the methods in this paper. A more complete description can be found elsewhere [14]. 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 block of a natural loop dominates all other blocks in the loop. Likewise, block B *postdominates* block A if all control paths from block A eventually lead to block B. A block always dominates and postdominates itself. We define the number of loop iterations as the number of times the header is executed once the loop is entered [11].

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 that can conditionally reach blocks containing different iteration branches. Figure 1 shows an algorithm to calculate the set of iteration branches I for a loop.

```
I = {}
DO
    FOR each block B in the loop L DO
        IF (B has two succs S1 and S2) AND (B ∉ I) THEN
            IF (S1 ∉ L) OR (S2 ∉ L) OR
               (S1 ∈ PostDom(Header(L))) OR
               (S2 ∈ PostDom(Header(L))) OR
               (there exists J,K ∈ I AND J ≠ K AND
                S1 ∈ PostDom(J) AND
                S2 ∈ PostDom(K)) THEN
              I = I ∪ B
WHILE (any change to I)
```

Figure 1: Finding the Set of Iteration Branches for a Loop

Figure 2(a) contains the code for a toy 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. (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 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 transfer to either block 3 or block 5, which have been identified as containing iteration branches.

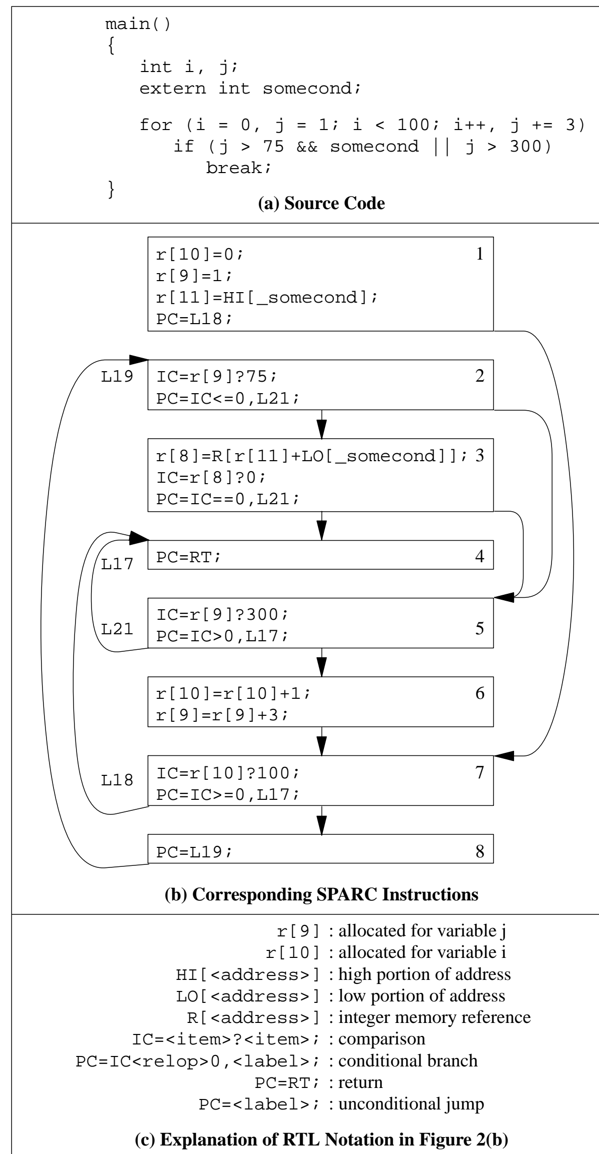


Figure 2: Example Loop with Multiple Exits

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*. The construction of the DAG can conceptually be accomplished by starting with the graph representing the loop, replacing all back edges with 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. Figure 3 shows the DAG depicting the precedence relationship between the blocks containing exit conditions from Figure 2.

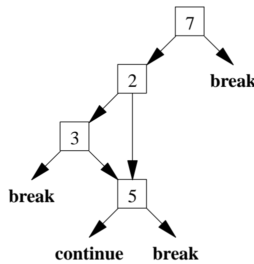


Figure 3: Precedence Relationship between Iteration Branches in Figure 2

## 2.2. 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 [1]. For each iteration branch we derive the information shown in Table 1. When all of the requirements listed in Table 1 are satisfied, the iteration branch is classified as *known*. Otherwise, the iteration branch is classified as *unknown*.

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 2 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.

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

In addition, various checks have to be made in case the iteration branch will always or never be satisfied. These checks depend on whether the *limit* is greater or less than the *initial* value, whether the sum of the *before* and *after* values are greater or less than zero, and the relational operator used in the comparison. Figure 4 shows two loops that require special checks. 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. 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) A Loop That Exits Immediately**

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

**(b) A Loop That May Never Exit**

Figure 4: Two Loops Requiring Special Checks

## 2.3. 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. We record this information as a range of iterations and attach a range to each node and edge. To calculate these ranges the DAG is processed in a preorder manner (i.e. all predecessors of a node are processed before the node is processed). 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 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 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

| Term | Explanation | Requirement |
|---|---|---|
| *variable* | The control variable on which the branch depends, which is the variable being compared in the block containing the iteration branch. | The control variable must be a basic induction variable, which is a variable `v` whose only assignments within the loop are of the form `v := v ± c` where `c` is a constant [14]. |
| *limit* | The value being compared to the *variable* in the block containing the branch. | The limit must be a constant. We will describe how this requirement can be relaxed in Section 3. |
| *relop* | The relational operator used to compare the *variable* and the *limit*. | Our initial description requires that the relational operator be an inequality operator (i.e. <, ≤, ≥, and >). We will describe how we relaxed this requirement in Subsection 2.5 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 a constant. We will describe how this requirement can be relaxed in Section 3. |
| *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 a constant and these constant 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 a constant and these constant changes must occur on each complete iteration of the loop. |
| *adjust* | An adjustment value of 0 or 1, which compensates for the difference between relational operators (e.g. < and ≤). | |

Table 1: Information Calculated for Each Iteration Branch

| branch | variable | register | limit | relop | initial | before | after | adjust | class | N |
|---|---|---|---|---|---|---|---|---|---|---|
| block 2 | j | r[9] | 75 | <= | 1 | 0 | 3 | 1 | 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 | 1 | known | 101 |
| block 7 | i | r[10] | 100 | >= | 0 | 0 | 1 | 0 | known | 101 |

Table 2: Derived Information for Each Iteration Branch in Figure 2

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 3's iteration branch is *unknown*. Thus, its two outgoing edges have ranges that match the range in node 3. 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).

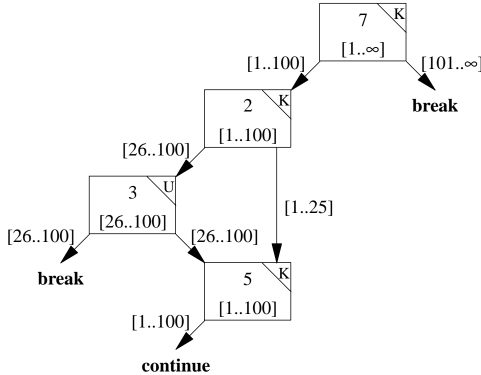


Figure 5: DAG of Branches with Ranges of Iterations

## 2.4. 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 a postorder manner (i.e. all successors of the node are processed before the node can be processed). 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 rules defined in Subsection 2.3.

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

[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 that has a transition to the loop header and is not in the loop.

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

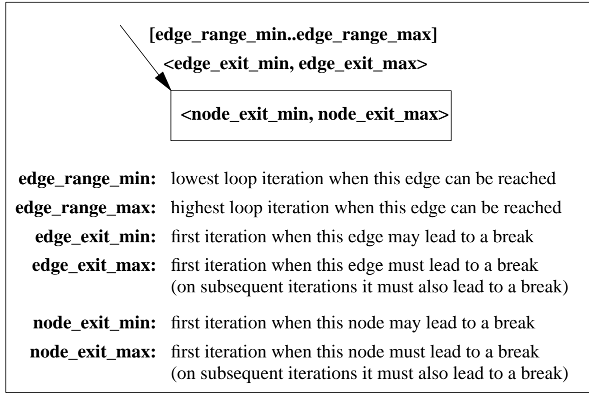


Figure 6: Notation Used in Rules
for Assigning Iteration Values

(1) If an edge is to a *break*, then both the *edge_exit_min* and *edge_exit_max* are assigned the value of *edge_range_min*. 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 to a *continue*, then the *edge_exit_min* and *edge_exit_max* values for that edge are marked as *unbounded*, which we will represent with '_'. (These transitions do not supply any information about when the loop exits.)

(3) Otherwise, the incoming edge is to a node representing an iteration branch and 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 3. 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*.

| Case | Condition | Test | Edge_Exit Assignment |
|---|---|---|---|
| 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 | — |

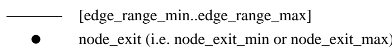


Table 3: Rules for Assigning
Iteration Values to an Incoming Edge

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

(1) 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.)

(2) 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*.)

(3) 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.)

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

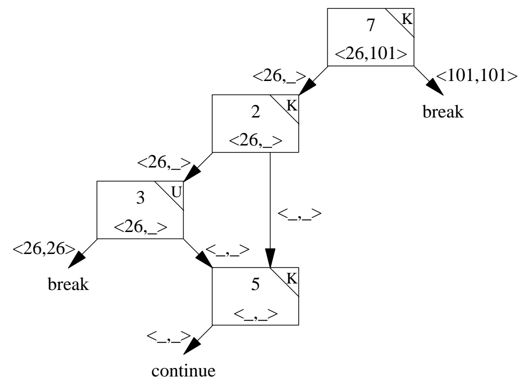


Figure 7: DAG of Iteration Branches
with Minimum and Maximum Iteration Values

### 2.5. Supporting Iteration Branches Using Equality Operators

As stated in Table 1, an iteration branch using an equality operator (i.e. == or !=) was initially described as always being treated as an *unknown* branch. 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 1 are satisfied. (1) First, every path ending in a back edge in the loop must include the iteration branch. Figure 8(b) shows 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).

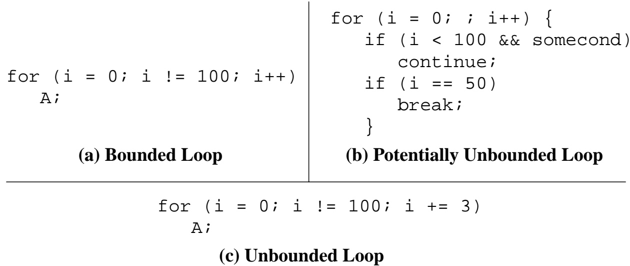


Figure 8: Examples of Loops with Iteration Branches Using Equality Operators

## 3. Supporting a Nonconstant 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 [5, 6] or as an assertion in the source code [7, 8]. 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 incorrectly specify the number of loop iterations.

All of the variables on which the number of loop iterations depend are frequently 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 defined in Subsection 2.2, but relax the requirement that the *limit* and *initial* values have to be constants. Figure 9 shows an example function and corresponding SPARC RTLs. (Some other 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 value 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
```

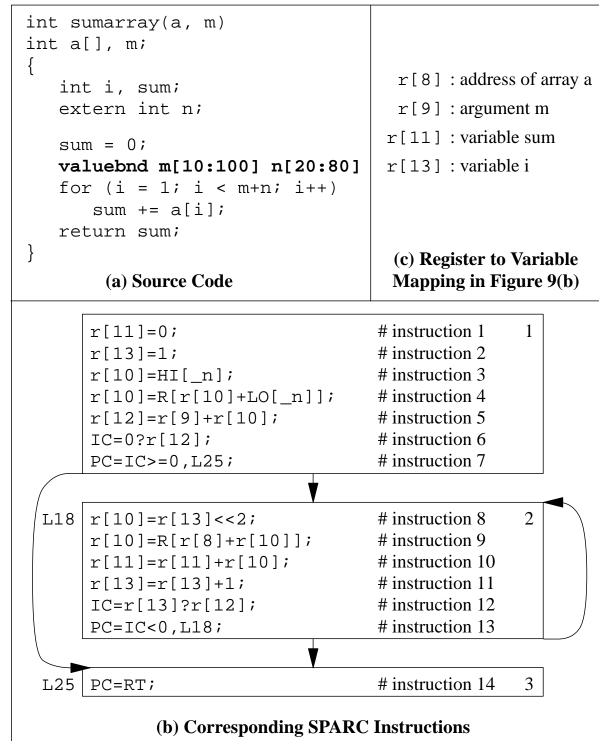


Figure 9: Loop with a Nonconstant Loop-Invariant Number of Iterations

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.

$$N = \left\lfloor \frac{limit - (initial + before) + adjust}{before + after} \right\rfloor + 1$$

$$= \left\lfloor \frac{m + n - (1 + 1)}{1 + 0} \right\rfloor + 0 + 1$$

$$= m + n - 1$$

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

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 intervention by the user is required. Note that the form of a value assertion is analogous to the form of timing constraint loop assertion that can be specified in the same environment [15].

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.

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

## 4. Supporting Variant Number of Iterations

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, sometimes the number of iterations depends upon values that can vary. For instance, the number of iterations of an inner loop often depends on the loop control variable of an outer loop. Consider the following for loops extracted from a sort program:

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

While the number of iterations of the inner loop is loop invariant with respect to the inner loop, the number of iterations varies depending on the value of the outer loop control variable. The number of iterations for the inner loop will range from 98..1. A naive assumption that the worst-case number of iterations for the inner loop is always 98 will result a significant overestimation when estimating the worst-case performance of the outer loop. Likewise, a significant underestimation of the best-case performance of the outer loop would result if the number of iterations of the inner loop is assumed to always be 1.

The average number of iterations of a inner loop with a single exit can be calculated when the difference between the number of loop iterations is incremented by a constant value each time the inner loop is entered for each iteration of the outer loop. Assume that:

(1) $f$ is the number of iterations of the inner loop on the first iteration of the outer loop,

(2) $d$ is the difference in the number of inner loop iterations for each successive iteration of the outer loop (note that $d$ may be negative),

(3) $n$ is the number of times that the outer loop iterates

The following equation represents the average of $n$ terms, where each term is the number of iterations of the inner loop on succeeding iterations of the outer loop. Note that $f + (n - 1)d$ is the number of iterations of the inner loop on the last iteration of the outer loop. The $d$ difference between the number of loop iterations can be guaranteed to be a constant when (1) only the *limit* or the *initial* value of the inner loop depends on an outer loop *variable* and (2) the *increment* (*before* + *after*) of the outer loop is an integral multiple of the *increment* of the inner loop. Note that the average number of iterations may still vary in best

and worst case since different values of $n$ may be used.

$$N_{avg}(f, d, n) = \frac{f + f + (n-1)d}{2}$$

We determine if any of the variables upon which the number of loop iterations depends is a basic induction variable for an outer level loop that encloses the current loop. This dependence could be from the *initial* value or the *limit* value of the current loop. Consider the source code and corresponding RTLs in Figure 11. In the example the *initial* value of the inner loop counter variable `r[10]` is found to be `r[11]+1`. The register `r[11]` is a basic induction variable for the outer loop. The compiler passes information about register `r[11]` to the timing analyzer concerning the number of iterations of the inner loop. This information includes the *initial* value (1), *limit* (99), and *increment* (1) of `r[11]`. In addition, the identification of the outer loop for which `r[11]` is an induction variable is also included. The timing analyzer determines that the number of iterations of the inner loop can vary from 98..1 since the initial value of the inner loop (`r[11]+1`) can vary from 2..99. Note the last value of `r[11]` at block 2 is 98 since after the increment in block 4 the outer loop is exited. Thus, the average number of iterations for the inner loop will be:

$$\frac{f + f + (n-1) * d}{2} = \frac{98 + 98 + (98 - 1) * (-1)}{2} = 49.5$$

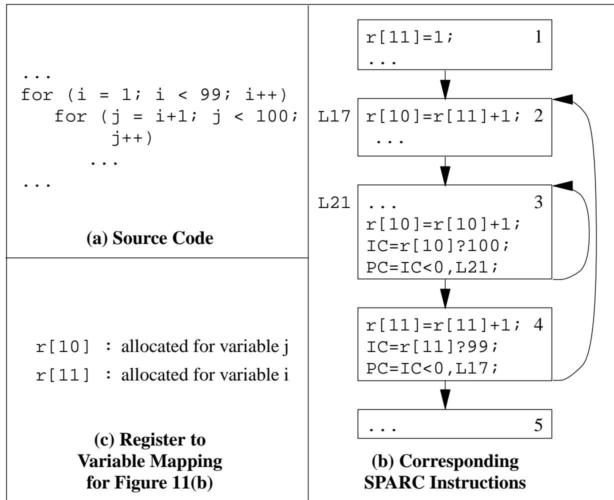


Figure 11: Number of Inner Loop Iterations Depends on the Outer Loop Counter Variable

When the timing analyzer performs loop analysis on a loop whose number of iterations can vary depending upon an outer loop induction variable, the analyzer will calculate two sets of times. One set is the conventional best and worst-case times. These times are calculated in case there is a timing constraint or request for the best or worst-case time for the inner loop. The other set is the average best and worst-case times using the average number of iterations. Note that when the average number of iterations is not an integer (49.5 in the example), the best-case average time for the loop will be calculated with the next lower integer (49 in the example) and the worst-case average time will be calculated with the next higher integer (50 in the example). The reason for the slightly conservative approach is that the loop analysis algorithm used by the timing analyzer is designed to work on an integral number of iterations [11, 12, 13]. These best and worst-case conventional and average times are passed up through the timing tree, which contains a node for each instance of a loop and function in the program. Included with the average times is an identification of the outer loop on which these average times depend. At the point the outer loop is encountered, the conventional times are abandoned and only the average times are used.

We enforced a couple of restrictions to ensure that an average number of iterations could be safely used in the timing analysis. First, we only use an average number of iterations when an inner loop is executed on every path of an outer loop. Figure 12 shows an outer loop with two conditionally executed inner loops. The number of iterations for the first inner loop varies from `99..1`. The number of iterations for the second loop varies from `1..99`. If the variable `somecond` is true for the first half of the outer loop iterations and false for the remaining half, then an underestimation will occur in the worst-case timing prediction if the average number of iterations were used for either inner loop. Second, we do not use the average times for an inner loop if it is nested within another loop for which we would calculate an average number of iterations. We have future plans to relax both of these restrictions so the number of iterations for more loops can be accurately bounded [16].

```
for (i = 0; i < 100; i++) {
   if (somecond)
      for (j = i+1; j < 100; j++)
         A;
   else
      for (j = 0; j < i; j++)
         A;
}
```

Figure 12: Example of Conditionally Executed Inner Loops

Given these restrictions, the average number of iterations can safely be used for the loop analysis in our timing analyzer. We always choose the worst possible execution time for a given iteration for our worst-case loop analysis. Due to the manner in which our timing analyzer handles cache misses, the predicted time for a given iteration can never be exceeded by the predicted time for the following

iteration of the loop (i.e. *predicted_time*(iteration $j$) ≥ *predicted_time*(iteration $j$+1)) [11, 12, 13]. Thus, the worst-case loop analysis using the average number of iterations will be safe since the execution times of the first $N_{avg}$ iterations will be at least as long as the execution times of any remaining iterations. An analogous argument can be made for best case.

Tables 4 and 5 show example programs that benefit from obtaining a more accurate estimation of loops whose number of iterations can vary. Note that the *Sort* program has been used in the past as one of the test programs to evaluate our timing analyzer [11, 12, 13]. These programs show the best and worst-case cycles required for executing with instruction caching and pipelining for the MicroSPARC I [17]. When using the average inner loop predictions, the predicted execution times were significantly tighter. 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. In worst case the number of iterations for each inner loop dependent on an outer loop variable was previously overestimated by about a factor of two. 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.

| Name | Description or Emphasis |
|---|---|
| Integ | Evaluates a Double Integral over a Trapezoidal Region |
| Interp | Polynomial Interpolation of 500 Points |
| Sort | Bubblesort of 500 Integers |
| Sym | Tests If a 500x500 Matrix Is Symmetric |

Table 4: Test Programs

| Best-Case Results | | | | | |
|---|---|---|---|---|---|
| Name | Observed Cycles | Previous Estimated Cycles | Prev. Ratio | Current Estimated Cycles | Curr. Ratio |
| Integ | 12,050,092 | 8,049,618 | 0.668 | 12,033,618 | 0.999 |
| Interp | 6,485,878 | 143,064 | 0.022 | 6,479,865 | 0.999 |
| Sort | 19,966 | 19,950 | 0.999 | 19,950 | 0.999 |
| Sym | 171 | 171 | 1.000 | 171 | 1.000 |
| **Worst-Case Results** | | | | | |
| Name | Observed Cycles | Previous Estimated Cycles | Prev. Ratio | Current Estimated Cycles | Curr. Ratio |
| Integ | 15,427,332 | 20,542,118 | 1.332 | 15,437,618 | 1.001 |
| Interp | 25,468,904 | 50,702,358 | 1.991 | 25,478,906 | 1.000 |
| Sort | 7,672,281 | 15,251,603 | 1.988 | 7,672,292 | 1.000 |
| Sym | 2,013,116 | 4,001,133 | 1.988 | 2,013,117 | 1.000 |

Table 5: Timing Analysis Results

## 5. Future Work

We plan to bound loop iterations for additional loops that our approach currently does not address. Occasionally, some loops have counter variables that are incremented by nonconstants. If the sequence of values used for the *increment* can be determined, then we may still be able to calculate a bounded number of iterations. Likewise, we will attempt to address multiple nested loops that are all dependent on the counter variables of outer loops.

Calculating the range of iterations when each block in a loop may be executed has other useful applications. This information may be used to modify our loop timing analysis to obtain tighter timing predictions since the timing analyzer would know in which iterations each path through a loop may be traversed. The fraction of time that a path within a loop may be executed can also be sometimes determined by examining the range of values for outer level loop counter variables when branches within that path depend on such variables. Thus, the approach of calculating an average time for a loop or a function may also be used to obtain tighter timing predictions even when the number of loop iterations do not vary. Finally, the range of iterations information for each block may be used by an optimizing compiler to eliminate nodes or transitions on infeasible paths and to restructure loops to produce more efficient code.

## 6. 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 13(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 nonconstant loop-invariant number of iterations is calculated when the variables on which the number of iterations depends cannot change values inside of the loop. Figure 13(b) depicts an example of this common type of loop. The user can specify the minimum and maximum values of these variables 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.

```
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 Loop-Invariant 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 13: Common Forms of Loops

Finally, timing analysis support is given to tightly predict the execution time of loops whose number of iterations is dependent on counter variables of outer level loops. These loops, such as the one shown in Figure 13(c), appear frequently in programs and can result in significant underestimations in best-case predictions and overestimations in worst-case predictions. Our approach more tightly predicts loops when the initial value or limit of the control variable in an inner loop depends on a control variable of an enclosing outer 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.

## 7. 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.

## 8. References

[1] M. E. Benitez and J. W. Davidson, "A Portable Global Optimizer and Linker," *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June 1988).

[2] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition,* Morgan Kaufmann, San Francisco, CA (1996).

[3] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pp. 318-328 (June 1988).

[4] H. S. Stone, *High-Performance Computer Architecture, Second Edition,* Addison Wesley, Reading, MA (1990).

[5] C. Y. Park and A. C. Shaw, "Experiments with a Program Timing Tool Based on a Source-Level Timing Schema," *Computer* **24**(5) pp. 48-57 (May 1991).

[6] Y. S. Li, S. Malik, and A. Wolfe, "Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software," *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, (December 1995).

[7] R. Chapman, A. Wellings, and A. Burns, "Integrated Program Proof and Worst Case Timing Analysis of SPARK Ada," *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, (June 1994).

[8] P. Puschner and C. Koza, "Calculating the Maximum Execution Time of Real-Time Programs," *Real-Time Systems* **1**(2) pp. 159-176 (September 1989).

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

[10] A. Ermedahl and J. Gustafsson, "Deriving Annotations for Tight Calculation of Execution Time," *Proceedings of European Conference on Parallel Processing*, pp. 1298-1307 (August 1997).

[11] R. Arnold, F. Mueller, D. Whalley, and M. Harmon, "Bounding Worst-Case Instruction Cache Performance," *Proceedings of the Fifteenth IEEE Real-Time Systems Symposium*, pp. 172-181 (December 1994).

[12] C. A. Healy, D. B. Whalley, and M. G. Harmon, "Integrating the Timing Analysis of Pipelining and Instruction Caching," *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, pp. 288-297 (December 1995).

[13] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon, "Timing Analysis for Data Caches and Set-Associative Caches," *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 192-202 (June 1997).

[14] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA (1986).

[15] L. Ko, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. Harmon, "Supporting the Specification and Analysis of Timing Constraints," *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 170-178 (June 1996).

[16] C. A. Healy, *Addressing Data Dependencies for Timing Analysis,* PhD Prospectus, Florida State University (April 1998).

[17] Texas Instruments, Inc., *Product Preview of the TMS390S10 Integrated SPARC Processor*, 1993.