Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints

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

Abstract

Predicting the worst-case execution time (WCET) of a real-time program is a challenging task. Though much progress has been made in obtaining tighter timing predictions by using techniques that model the architectural features of a machine, significant overestimations of WCET can still occur. Even with perfect architectural modeling, dependencies on data values can constrain the outcome of conditional branches and the corresponding set of paths that can be taken in a program. While value-dependent constraint information has been used in the past by some timing analyzers, it has typically been specified manually, which is both tedious and error prone. This paper describes efficient techniques for automatically detecting value-dependent constraints by a compiler and automatically exploiting these constraints within a timing analyzer. The result is tighter timing analysis predictions without requiring additional interaction with a user.

1. Introduction

Obtaining accurate worst-case execution time (WCET) predictions of programs is a challenging task. Various features of the architecture, such as caches and pipelines, can affect the execution time of an instruction and these features need to be modeled while analyzing the control flow of a program. However, even with perfect architectural modeling, significant overestimations of WCET can still occur since dependencies on data values can constrain the outcome of conditional branches and restrict the set of paths that can be taken. While value-dependent constraint information has been used in the past by some timing analyzers, it has typically been specified manually, which is both tedious and error prone. This paper describes how value-dependent constraints can be automatically detected by a compiler and exploited by a timing analyzer.

2. Related Work

Some constraint-based timing analyzers use value-dependent constraints to obtain more accurate estimations of execution time. Li *et al*. performed timing analysis using an Implicit Path Enumeration (IPE) technique [1], which uses integer linear programming (ILP) to solve constraints about the program to obtain timing predictions. Their technique automatically calculates *program structural constraints* from the program control flow graph and used value-dependent constraints, which they called *program functionality constraints*. The work of Ottosson and Sjödin [2] extended the IPE technique by using finite domain constraints to model the architectural features of the hardware. However, in both approaches these value-dependent constraints were entered manually by the user, which is both a tedious and error-prone task.

Recent work by Ermedahl and Gustafsson [3] and by Lundqvist and Stenström [4] use abstract interpretation and symbolic execution to automatically derive many value-dependent constraints. These approaches are quite powerful, but effectively requires simulating all paths of a loop for every loop iteration. Thus, these approaches require significant analysis overhead, which would be undesirable when analyzing long running programs.

Another type of value-dependent constraint is the number of loop iterations. We have implemented techniques to automatically determine the minimum and maximum iterations for many loops with multiple exit conditions and loops whose number of iterations depend on loop-invariant variables or counter variables of outer loops [5]. The abstract interpretation and symbolic execution approaches [3, 4] also provide a more powerful and less efficient method to calculate bounds on loop iterations. In this paper, we address detecting and exploiting value-dependent constraints that constrain execution paths rather than the number of iterations that a loop can execute.

3. Automatic Detection of Constraints

A value-dependent constraint causes the outcome of a conditional branch to be known under certain conditions. We implemented techniques to detect these constraints, which we classified as *effect-based* and *iteration-based*.

3.1. Detecting Effect-Based Constraints

The compiler performs analysis to determine if the outcome of a conditional branch is known at any given point in the control flow. First, the compiler calculates the set of registers and variables upon which a branch (and its associated comparison) depends. This set is calculated by expanding the effects of the comparison instruction associated with the branch. For instance, consider the SPARC instructions represented as RTLs (Register Transfer Lists) and the associated expanded comparison in Figure 1. A comparison is expanded by searching backwards for assignments to registers in the comparison until all registers are replaced or the beginning of a block with multiple predecessors is encountered. Loop-invariant registers in the expression are expanded from the preheader of the loop in which they are assigned values. Next, the compiler determines the set of effects associated with assignments to registers and variables for each basic block. Each branch is examined to see if it could be affected by the block. Thus, the compiler can determine that a basic block updating the global variable `g` could affect the result of the branch in Figure 1. Updates to the registers `r[1]` (`%g1`) or `r[8]` (`%o0`) would have no effect.

Instructions in a Basic Block

```
r[1]=HI[_g];          /* sethi %hi(_g),%g1       */
r[8]=R[r[1]+LO[_g]]; /* ld    [%g1+%lo(_g)],%o0 */
IC=r[8]?5;            /* cmp   %o0,5             */
PC=IC<0,L20;          /* bl    L20               */
```

Expanded Comparison

```
IC=R[HI[_g]+LO[_g]]?5;
```

Figure 1: Example of Expanding a Comparison

A state is associated with each conditional branch, which can have one of three values: *unknown*, *fall-through*, or *jump*. The authors determine if a branch becomes known by substituting the value assigned for the variable or register and evaluating the expanded comparison in the compiler. The compiler issues a directive to the timing analyzer for each branch placed in an *unknown*, *fall-through*, or *jump* state by an effect in the block. Thus, this analysis requires $O(B*C)$ complexity, where B is the number of basic blocks and C is the number of conditional branches. A more complete explanation for detecting branch states has been described in previous work [6].

Consider the source code in Figure 2(a). The corresponding control flow that is generated by the compiler is shown in Figure 2(b). While the control flow in the figure is represented at the source code level, the analysis is performed by the compiler at the machine instruction level after compiler optimizations are applied to provide more accurate timing predictions. Note that some branches in Figure 2(b) have conditions that are reversed from the code in Figure 2(a) to depict the branch conditions that are evaluated at the machine instruction level. Only when the condition associated with a branch in a block is evaluated to be true will the jump (**J**) occur. If the condition is not true, then control will fall (**F**) into the next sequential block. The control flow also shows the effect-based constraints, which are enclosed in curly braces and associated with basic blocks or control-flow transitions. Figure 2(c) describes the explicit value-dependent constraints that are automatically detected by the compiler and passed to a timing analyzer. The initialization of `i` in block 1 (`i=0;`) puts the branch in block 2 (`a[i]!=0`) in an *unknown* state (**2U**) and the branch in block 9 (`i<1000`) in a *jump* state (**9J**). In addition, the assignments to `odd` in blocks 1 and 5 (`odd=0;`) and in block 6 (`odd=1;`) cause the branch in block 4 (`odd==0`) to *jump* (**4J**) and *fall through* (**4F**), respectively. Likewise, the assignment to `quit` in blocks 1 (`quit=0;`) and 3 (`quit=1;`) cause the branch in block 8 (`quit!=0`) to *fall through* (**8F**) and *jump* (**8J**), respectively. Finally, the increment of `i` in block 7 (`i++;`) sets the states of the branches in blocks 2 (`a[i]!=0`) and 9 (`i<1000`) to *unknown* (**2U**,**9U**) since they depend on the value of `i`.

```
sumodd = sumeven = 0;
odd = quit = 0;
for (i = 0; !quit &&
     i < 1000; i++)
   if (a[i] == 0)
      quit = 1;
   else if (odd) {
      sumodd += a[i];
      odd = 0;
      }
   else {
      sumeven += a[i];
      odd = 1;
      }
```

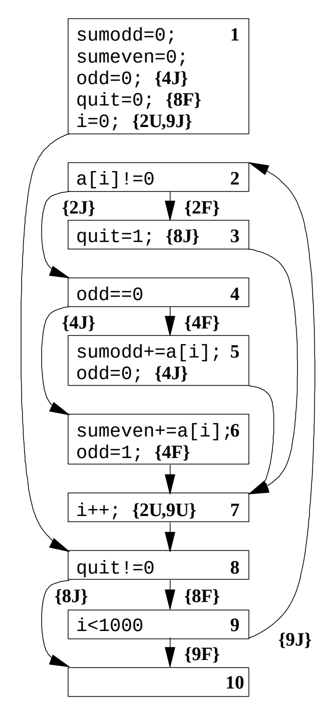
(a) Source Code

(1) blk 1 nullifies blk 2
(2) blk 1 makes blk 4 jump
(3) blk 1 makes blk 8 fall thru
(4) blk 1 makes blk 9 jump
(5) blk 3 makes blk 8 jump
(6) blk 5 makes blk 4 jump
(7) blk 6 makes blk 4 fall thru
(8) blk 7 nullifies blks 2,9

(c) Explicit Constraints

(1) 8
(2) 8→9
(3) 8→9→2→3→7
(4) 8→9→2→4→5→7
(5) 8→9→2→4→6→7

(d) Paths in Loop

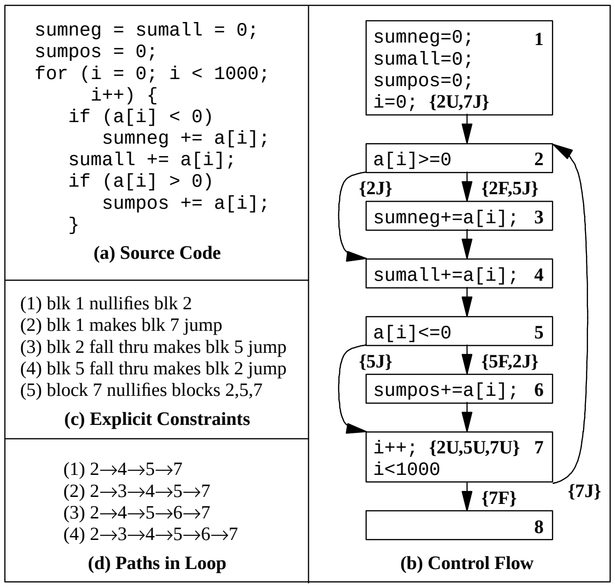


(b) Control Flow

Figure 2: Effects of Assignments on Branches

Figure 2(b) also shows implicit value-dependent constraints. When a branch has a given outcome, then it will have the same outcome again unless the variables or registers being compared are affected. Thus, a fall-through (**F**) or jump (**J**) transition from a branch will implicitly cause that same branch to be in a *fall-through* or *jump* state, respectively. These implicit constraints are not explicitly passed to a timing analyzer since a timing analyzer can create them when it is performing analysis on paths.

The source code in Figure 3(a) and corresponding control flow in Figure 3(b) depict a situation where one conditional branch may be logically correlated with another branch. In other words, the direction taken by one conditional branch may indicate the direction taken by another conditional branch. If block 2 (`a[i]>=0`) falls into block 3, then the value of `a[i]` is negative and block 5 (`a[i]<=0`) must jump to block 7 (**5J**). This is described by value-dependent constraint 3 in Figure 3(c). Note that if block 2 (`a[i]>=0`) jumps to block 4, there is no guarantee that block 5 (`a[i]<=0`) will fall through to block 6 since the value of `a[i]` could have been zero. The compiler evaluates each pair of branches in a function to determine if there is a logical correlation between one branch and another. Thus, this analysis requires $O(C^2)$ complexity, where C is the number of conditional branches. Note that a branch is always logically correlated with itself and these self correlations are implicit constraints. The exact conditions when one branch is logically correlated with another have been described in previous work [6].



```
sumneg = sumall = 0;
sumpos = 0;
for (i = 0; i < 1000;
     i++) {
   if (a[i] < 0)
      sumneg += a[i];
   sumall += a[i];
   if (a[i] > 0)
      sumpos += a[i];
   }
```

(a) Source Code

(1) blk 1 nullifies blk 2
(2) blk 1 makes blk 7 jump
(3) blk 2 fall thru makes blk 5 jump
(4) blk 5 fall thru makes blk 2 jump
(5) block 7 nullifies blocks 2,5,7

(c) Explicit Constraints

(1) $2 \rightarrow 4 \rightarrow 5 \rightarrow 7$
(2) $2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7$
(3) $2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$
(4) $2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$

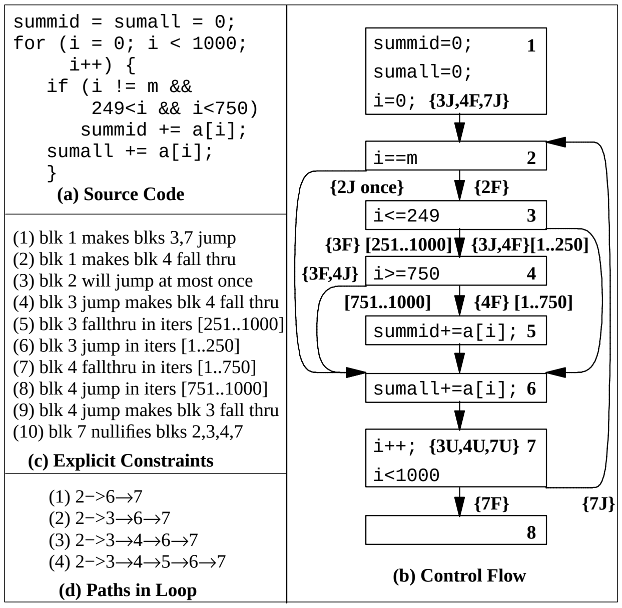
(d) Paths in Loop

Figure 3: Logical Correlation between Branches

3.2. Detecting Iteration-Based Constraints

A basic induction variable is a variable or register that is incremented or decremented by a constant value on each iteration of a loop. Some branches compare a basic induction variable to a constant. In these situations, the compiler can determine the ranges of iterations in which such a branch will fall through or jump. The compiler produces directives for a timing analyzer that indicate ranges of iterations for each of the two outgoing edges of the block containing the branch. The manner in which this information is derived is described elsewhere [5].

Consider the source code and corresponding control flow shown in Figures 4(a) and 4(b). While `i` can range from 0..999 as each path in the loop is entered, the number of corresponding iterations in the loop will range from 1..1000. Thus, the compiler associates ranges of iterations with transitions from blocks that compare basic induction variables to constants. For instance, block 3 (`i<=249`) will only fall through to block 4 when the loop is performing the last 750 iterations (**[251..1000]**). Constraints 5-8 in Figure 4(c) depict the range of iterations when various transitions in the loop can be taken. An implicit iteration-based constraint is that the header of the loop (block 2 in Figure 4(b)) can be executed in every loop iteration (**[1..1000]** for Figure 4). Sometimes a basic induction variable is compared to nonconstant loop invariant values, as shown in block 2 (`i==m`) of Figure 4(b). The value of `m` is not known, but it is invariant with respect to the loop. When the comparison of such a branch is an equality test (== or !=), then the transition that occurs when the two values are equal can take place at most once for each execution of the loop since the basic induction variable changes by a constant value on each iteration. Constraint 3 in Figure 4(c) shows that the compiler determines that block 2 will jump to block 6 at most once (**2J once**). The analysis to detect iteration-based constraints requires $O(C)$ complexity, where C is the number of conditional branches, since each branch must be inspected once.



```
summid = sumall = 0;
for (i = 0; i < 1000;
     i++) {
   if (i != m &&
       249<i && i<750)
      summid += a[i];
   sumall += a[i];
   }
```

(a) Source Code

(1) blk 1 makes blks 3,7 jump
(2) blk 1 makes blk 4 fall thru
(3) blk 2 will jump at most once
(4) blk 3 jump makes blk 4 fall thru
(5) blk 3 fallthru in iters [251..1000]
(6) blk 3 jump in iters [1..250]
(7) blk 4 fallthru in iters [1..750]
(8) blk 4 jump in iters [751..1000]
(9) blk 4 jump makes blk 3 fall thru
(10) blk 7 nullifies blks 2,3,4,7

(c) Explicit Constraints

(1) $2 -> 6 \rightarrow 7$
(2) $2 -> 3 \rightarrow 6 \rightarrow 7$
(3) $2 -> 3 \rightarrow 4 \rightarrow 6 \rightarrow 7$
(4) $2 -> 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$

(d) Paths in Loop

Figure 4: Ranges of Iterations and Branch Outcomes

4. Using Constraints in a Timing Analyzer

The analysis techniques described in the previous section to identify value-dependent constraints could be used by a variety of timing analyzers, which include those that use an integer linear programming (ILP) solver. While an ILP approach can be simple, elegant, and quite powerful, there are a few disadvantages. For instance, an ILP approach works best when each basic block can be associated with a single time, which allows this time to be expressed as a constraint associated with that block. Caching and pipelining change the context in which a block could be executed and can often affect its associated execution time. While approaches have been suggested for addressing caching behavior [1], it is still unclear how pipelining can be accurately modeled across multiple blocks. More importantly, the time required for the analysis with an ILP approach has worst-case exponential complexity. A program that required only a few seconds of timing analysis using a more traditional approach [7] required minutes using an ILP approach [1]. In fact, ILP methods can be used to solve many compiler optimization problems, but are infrequently used in production compilers due to potentially excessive compilation time. Thus, the authors decided it would be worthwhile to investigate how value-dependent constraints could be exploited by a non-ILP based timing analyzer.

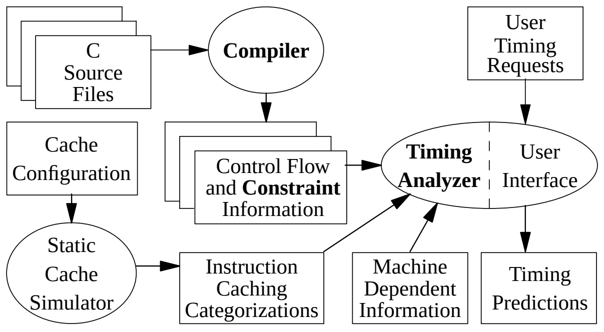


Figure 5: Overview of Timing Analysis Process

Figure 5 depicts the overall organization of the non-ILP timing analysis environment that was modified to exploit value-dependent constraint information. An optimizing compiler [8] was used to produce control flow and value-dependent constraint information as a side effect of the compilation of a file [9, 5]. A static cache simulator uses the control flow information to construct a control-flow graph of the program that consists of the call graph and the control flow of each function. The program control-flow graph is then analyzed and a caching categorization for each instruction in the program is produced [9]. Next, a timing analyzer uses the control flow and constraint information, caching categorizations, and machine dependent information (e.g. pipeline characteristics) as input to make timing predictions [9, 7]. Finally, a graphical user interface is invoked that allows the user to request predictions for portions of the program [10].

4.1. Generating Path Constraints

The timing analyzer uses the value-dependent constraints to calculate a minimum and maximum number of iterations associated with each path during the execution of a loop. Table 1 depicts information associated with each loop path described in Figures 2(d), 3(d), and 4(d). The total number of loop iterations is automatically calculated using techniques described in previous work [5]. A path is a sequence of blocks in a loop connected by control-flow transitions. Each path starts with the loop header. *Exit* paths are terminated by a block with a transition out of the loop. *Continue* paths are terminated by a block with a transition to the loop header. The next two columns indicate the range of possible and unique iterations associated with each path. The final two columns show the minimum and maximum number of times the path could be executed in the loop.

Figure 6 gives a high-level description of the algorithm used to calculate the information given in the last four columns of Table 1. The remainder of this section provides examples to illustrate how this information is calculated. Except for the construction of the REACH_SELF table, the complexity of the algorithm is $O(P^2)$, where P is the number of paths in the loop. In practice, the construction of the REACH_SELF table was not time consuming since we found that most paths in a loop could either immediately follow themselves or could only exit the loop.

Effect-based constraints are associated with a block or a transition between blocks. For each path in a loop the timing analyzer traverses the basic blocks and transitions between blocks in the order in which the path would be executed. When an effect-based constraint is encountered, it is added to a list of constraints for that path. If another effect-based constraint is later encountered for that same branch, then the current constraint is nullified.

Effect-based constraints can be used to detect infeasible paths. Figure 7 depicts the constraints being propagated through path 4 in Figure 3(d). The transition from block 2 to block 3 causes the branch in block 5 to be placed in a *jump* state (**5J**). The branch in block 5 is encountered with this constraint (**5J**) still in effect and the transition from block 5 to block 6 in path 4 is deemed illegal. When such an infeasible path is encountered, the timing analyzer removes the path to prevent any additional analysis time to be spent on it.

Loop	Total Iters	Path ID	Exit Path	Continue Path	Possible Iterations	Unique Iterations	Minimum Iterations	Maximum Iterations
Loop in Figure 2	1,001	1	Y	N	[1001..1001]	∅	0	1
		2	Y	N	[1001..1001]	∅	0	1
		3	N	Y	[1000..1000]	∅	0	1
		4	N	Y	[2..1000]	∅	0	500
		5	N	Y	[1..1000]	∅	0	500
Loop in Figure 3	1,000	1	Y	Y	[1..1000]	∅	0	1,000
		2	Y	Y	[1..1000]	∅	0	1,000
		3	Y	Y	[1..1000]	∅	0	1,000
		4	N/A	N/A	N/A	N/A	N/A	N/A
Loop in Figure 4	1,000	1	Y	Y	[1..1000]	∅	0	1
		2	N	Y	[1..250]	[1..250]-1	249	250
		3	Y	Y	[751..1000]	[751..1000]-1	249	250
		4	N	Y	[251..750]	[251..750]-1	499	500

Table 1: Information for Each Path in Figures 2(d), 3(d), and 4(d)

```
/* remove infeasible paths */
FOR each path P in the loop DO
   Propagate effect-based constraints in P.
   IF any transition in P is not feasible THEN
      Remove P from the loop.

/* calculate CAN_FOLLOW table using effect-based constraints */
FOR each path P in the loop DO
   IF P is a continue path THEN
      FOR each path Q in the loop DO
         Propagate effect-based constraints
            at end of P through Q.
         IF any infeasible transition in Q THEN
            CAN_FOLLOW[P][Q] = FALSE.
         ELSE
            CAN_FOLLOW[P][Q] = TRUE.
   ELSE
      FOR each path Q in the loop DO
         CAN_FOLLOW[P][Q] = FALSE.

/* calculate REACH_SELF table using CAN_FOLLOW table */
FOR each path P in the loop DO
   IF CANFOLLOW[P][P] THEN
      REACH_SELF[P] = 1.
   ELSIF P is not a continue path THEN
      REACH_SELF[P] = 0.
   ELSE
      Recursively inspect the CAN_FOLLOW table
      to determine the shortest number of paths
      to be traversed before P can be reached.
      Zero represents P cannot reach itself.

/* process once constraints */
FOR each path P in the loop DO
   IF a once constraint was found on
      a transition in P THEN
      P->once = TRUE.
   ELSE
      P->once = FALSE.
   P->nonuniqiters = 0.
   FOR each block B in P DO
      IF B’s other outgoing transition has a
         once constraint THEN
         P->nonuniqiters += 1.

/* initialize possible iteration path information, where N
   represents the total loop iterations */
FOR each path P in the loop DO
   P->range = ∅.
   IF P is a continue path THEN
      P->range = P->range ∪ [1..N-1].
   IF P is an exit path THEN
      P->range = P->range ∪ [N..N].

/* constrain possible iterations using iteration-based constraints */
FOR each path P in the loop DO
   Propagate iteration-based constraints in P.
   P->range = P->range ∩
            iteration range at end of P.
   IF P->range = ∅ THEN
      Remove P from the loop.

/* constrain iterations of each path that cannot reach itself */
Construct a DAG D representing the execution
   order of paths P where REACH_SELF[P] == 0.
FOR each nonleaf path P in D, where P is not
    processed until all paths it can reach
    are processed DO
   S = first immediate successor of P.
   P->range.low = S->range.low - 1.
   P->range.high = S->range.high - 1.
   FOR each remaining path S that is an
       immediate successor of P in D DO
      IF S->range.low - 1 < P->range.low THEN
         P->range.low = S->range.low - 1.
      IF S->range.high - 1 > P->range.high THEN
         P->range.high = S->range.high - 1.

/* calculate unique iterations for each path */
FOR each path P in the loop DO
   P->uniqrange = P->range
   FOR each path Q, where Q≠ P D O
      P->uniqrange = P->uniqrange− Q->range.

/* assign minimum number of iterations for each path */
FOR each path P in the loop DO
   P->miniter =
      number of iterations in P->uniqrange.
   P->miniter -= P->nonuniqiters.

/* assign maximum number of iterations for each path */
FOR each path P in the loop DO
   IF REACH_SELF[P] = 0 O R P->once THEN
      P->maxiter = 1.
   ELSE
      P->maxiter =
         number of iterations in P->range.
      IF REACH_SELF[P] > 1 THEN
         P->maxiter =
            ceil(P->maxiter/REACH_SELF[P]).

/* assign each path to a set of paths */
s = 0.
FOR each path P in the loop DO
   IF P->range ∩ with existing set i THEN
      P->set = i;
   ELSE
      P->set = ++s;
```

Figure 6: Algorithm for Calculating Path Iteration Information in Table 1

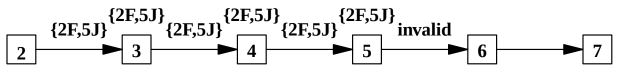


Figure 7: Path 4 in Figure 3(d) Is Not Feasible

The maximum number of iterations for a path can sometimes be constrained by effect-based constraints. Consider paths 1 and 2 in Figure 2(d), which are *exit* paths because they end with a transition to block 10 that is outside the loop. Value dependent constraint 5 in Figure 2(c) indicates that when block 3 (`quit=1;`) in Figure 2(b) is executed, block 8 (`quit!=0`) will jump to block 10. When the timing analyzer detects that an effect-based constraint can reach the end of the path without nullification, the timing analyzer propagates the constraint through all the paths of the loop to see if it can reach the branch identified in the constraint. Figure 8 illustrates that the constraint causing the branch in block 8 to *jump* (**8J**) reaches the end of path 3 and that paths 2, 3, 4, and 5 cannot follow path 3 since they require a fall through from block 8 to block 9. Figure 9 shows that the constraint for branch 4 reaching the end of paths 4 and 5 from Figure 2 contains the opposite outcome of branch 4 in the same path. This causes these paths not to be taken on the next loop iteration. Finally, data flow analysis was used to determine which effect-based constraints outside the loop are guaranteed to reach the entry of the loop. First instance, path 4 of Figure 2 cannot be executed on the first iteration due to constraint {**4J**} reaching block 4 from block 1.

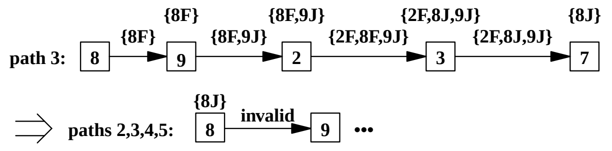


Figure 8: Paths 2-5 Cannot Follow Path 3 in Figure 2(d)

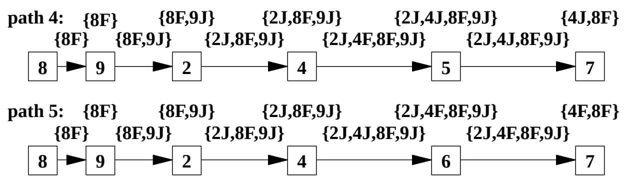


Figure 9: Paths 4 and 5 Cannot Immediately Follow the Same Path in Figure 2(d)

A *Can Follow* matrix is constructed by the timing analyzer that indicates for each path the set of paths that can legally follow it on the next iteration. If a constraint from one path can reach its associated branch in other paths without being nullified, then such paths that have transitions that do not satisfy the constraint are marked as illegal in the matrix. No paths are allowed to follow a path that only exits. Table 2 depicts the matrix of paths that can legally follow each path in Figure 2(d).

Current Path in Loop	Paths That Can Immediately Follow				
	1	2	3	4	5
1	N	N	N	N	N
2	N	N	N	N	N
3	Y	N	N	N	N
4	N	Y	Y	N	Y
5	N	Y	Y	Y	N

Table 2: Can Follow Matrix for Figure 2

After the matrix is completed, it is examined to see if restrictions on the number of iterations associated with each path can be applied. In general, the timing analyzer examines the matrix for each path to determine the fewest number of other paths required to be traversed before the current path can be executed again. If the algorithm indicates that a path cannot reach itself, then the path will be assigned a maximum of one iteration. Paths 1, 2, and 3 of Figure 2(d) are all assigned a maximum number of one iteration because they cannot reach themselves after executing. If a path cannot directly follow itself, but can eventually be reached again, then it cannot execute on every iteration of the loop. If the algorithm indicates that the *K* iterations required to be executed before a *continue* path can reach itself is greater than one, then it is assigned a maximum number of iterations from *ceil*(*R*/*K*), where *R* is the possible number of iterations for the path. Paths 4 and 5 of Figure 2(d) can only execute again on the second iteration after it last executed. Thus, paths 4 and 5 are assigned *ceil*(999/2) and *ceil*(1,000/2), respectively, or 500 maximum iterations.

The maximum number of iterations can sometimes be constrained by analyzing iteration-based constraints. The header block is assigned a range that spans all iterations of the loop. This range is propagated through each path. When a transition is encountered that has an iteration-based constraint, the range in the constraint is intersected with the range in the current block in the path. Figure 10 illustrates how iteration-based constraints are propagated through path 4 in Figure 4(d). The transition from block 3 (`i<=249`) to block 4 results in the range [1..1000] being intersected with [251..1000], which is the range specified in constraint 5 of Figure 4(c). The transition from block 4 (`i>=750`) to block 5 results in the current range of [251..1000] being intersected with [1..750]. Thus, path 4 can only possibly execute in iterations [251..750].

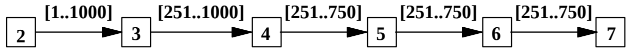


Figure 10: Iteration-Based Constraints Propagated Through Path 4 in Figure 4

If a path can only be executed in a given range of iterations, then the maximum iterations in which that path can execute cannot be greater than the number of iterations in the range. A path with no possible iterations is infeasible and is removed from the list of paths by the timing analyzer. Note that the range of a path that only exits is always the last iteration of the loop, which is the case for paths 1 and 2 of Figure 2(d). Likewise, if path A cannot reach itself and can only be immediately followed by a different path B, which has a range [Bmin..Bmax], then path A's range cannot span more than [Bmin-1..Bmax-1]. For instance, Table 2 shows that path 3 of Figure 2(d) always leads to path 1, which has an iteration range of [1001..1001]. Thus, path 3's possible range of iterations is [1001-1..1001-1] or [1000..1000] for WCET analysis.

The minimum number of iterations of a path is calculated by simply subtracting the possible range of iterations of all other paths in the loop from the possible range of iterations for the current path. The result is the unique set of iterations for the current path, which is the minimum number of times that the path has to execute. There is one exception to this rule. Consider path 1 in Figure 4(d). Its maximum number of iterations is one due to constraint 3 (**2J once**) in Figure 4(c). We do not reduce the range of unique iterations of the other paths, but do indicate that one iteration in these paths may not be unique.

4.2. Using the Constraints in Loop Analysis

The authors decided to use the minimum and maximum iterations associated with each loop path to obtain tighter WCET loop predictions without restricting the order in which these paths are evaluated. There were several reasons why this approach was used. First, our approach supports paths that can execute at most once, but in any iteration. Consider path 1 of the loop in Figure 4. This situation may occur frequently in numerical applications. For instance, special conditions are often checked for the diagonal elements of a matrix (diagonal systems). Second, our approach deals with paths that have dependencies on other paths, such as paths 4 and 5 in Figure 2. Finally, our timing analyzer often calculates an average WCET for a loop using an average number of iterations when the number of iterations can vary depending on the value of a outer loop counter variable [5]. Using our approach allows the calculation of a safe average WCET since the longest paths are selected first in our loop analysis algorithm.

In addition, we calculate sets of paths, where the range of iterations of the paths in one set do not overlap with other sets. Each path is assigned to a single set of paths, We use the maximum number of iterations that can be executed by a set of paths, which is the number of iterations in the set's range. Table 3 depicts an example with 4 paths and 2 sets. Each set of paths can only execute a maximum of 50 iterations. If only the maximum iterations of each path was used, then two paths from a single set could be selected and a significant overestimation may occur when the paths in one set require many more cycles than the paths in the other set. Our approach has limitations. Consider if a fifth path existed in this example which could execute in any iteration of the loop. All of the loop paths would be assigned to a single set, which could result in an conservative timing prediction. Fortunately, inequality tests (<, <=, >=, >) on loop induction variables do not occur frequently.

Path	Possible Iterations	Min Iters	Max Iters	Set
1	[1..50]	0	50	1
2	[1..50]	0	50	1
3	[51..100]	0	50	2
4	[51..100]	0	50	2

Table 3: Example Illustrating Use of Path Sets

Figure 11 shows how the WCET loop analysis algorithm uses this information. Let *N* be the maximum number of iterations and *P* be the number of paths in a loop. The DO-WHILE will process at most the minimum of *N* or 2*P* total iterations since the first misses and first hits in each path can miss or hit at most once, respectively.[1]

The algorithm selects the longest path on each iteration of the loop from the set of paths that can still possibly execute. In order to demonstrate the correctness of the algorithm, one must show that no other path for a given iteration of the loop will produce a longer worst-case time than that path selected by the algorithm. Descriptions of how the caching categorizations and pipeline information are used in the loop analysis and correctness arguments about selecting the longest path using these categorizations and information have been given in previous work [9, 7]. Thus, it remains to be shown that each time a path is selected, it is really chosen from the paths that can still possibly execute given that the minimum and maximum number of iterations for each path and set were accurately estimated. A path's number of required iterations is its minimum iterations to be performed. The nonrequired iterations of a path is the difference between its maximum and minimum number of iterations. A path is initially chosen in the IF-THEN-ELSE construct at the beginning of the DO-WHILE loop in Figure 11. If the iterations remaining is greater than the required iterations left to be

[1] If the number of paths within a loop exceeds a reasonable limit, then the loop control flow is partitioned to reduce the timing analysis complexity [11].

```
/* calculate required and nonrequired path information */
req_iters = 0.
FOR P = each path in the loop DO
   P->req_iters = P->min_iters.
   P->nonreq_iters =
      P->max_iters - P->min_iters.
   req_iters += P->min_iters.
nonreq_iters = $N$ - req_iters.
/* process all iterations of the loop */
iters_handled = 0.
pipeline_info = NULL.
WHILE iters_handled < $N$ DO
   /* process iters while longest path has a first miss or first hit */
   DO
      IF req_iters < $N$ - iters_handled THEN
         Find longest path P where
            P->req_iters+P->nonreq_iters > 0 &&
            P->set.maxiters > 0.
      ELSE
         Find longest path P where
            P->req_iters > 0 &&
            P->set.maxiters > 0.
      Concatenate pipeline_info with the current
         worst-case union of executable paths.
      iters_handled += 1.
      IF P->req_iters > 0 THEN
         P->req_iters -= 1.
         req_iters -= 1.
      ELSE
         P->nonreq_iters -= 1.
         nonreq_iters -= 1.
      P->set.maxiters -= 1.
   WHILE encountered a first miss or first hit
         AND iters_handled < $N$
   /* Efficiently process iterations for the current longest path */
   IF iters_handled < $N$ THEN
      nonreq_iters_to_do =
         min(nonreq_iters, P->nonreq_iters,
             P->set.maxiters - P->req_iters).
      iters_to_do =
         P->req_iters + nonreq_iters_to_do.
      req_iters -= P->req_iters.
      nonreq_iters -= nonreq_iters_to_do.
      P->set.maxiters -= iters_to_do.
      P->req_iters = 0.
      P->nonreq_iters -= nonreq_iters_to_do.
      Concatenate pipeline_info iters_to_do
         times with current worst-case union.
      iters_handled += iters_to_do.
```

Figure 11: WCET Loop Analysis Algorithm

processed (sum of each path's minimum iterations not yet processed), then the path selected is chosen from any path that has any iterations that can be performed. Otherwise, the iterations remaining must be equal to the required loop iterations remaining and the path must be selected only from paths that have remaining required iterations left. The code after the DO-WHILE in the algorithm efficiently uses repeated instances of a path that has no first misses or first hits and thus will remain the longest path since its worst-case behavior cannot change. This code processes the remaining required iterations of the path and the minimum of the remaining nonrequired iterations of the path, the set of paths to which the path belongs, or the entire loop. Therefore, the paths that can still possibly execute is accurate since a given path's required iterations are always processed before its nonrequired iterations and the number of nonrequired iterations to be processed for a path is never allowed to exceed the number of nonrequired iterations remaining in the loop.

5. Results

Table 4 depicts programs where the worst-case paths were constrained by dependencies on data values to evaluate the effectiveness of detecting and exploiting value-dependent constraints. The *Sumoddeven*, *Sumnegpos*, and *Summidall* programs correspond to the examples illustrated in Figures 2, 3, and 4, respectively. The *Expint* program performs more computation when a loop variable is equal to a loop-invariant value on a single loop iteration. The *Frenel* program takes different paths on the odd and even steps in the evaluation of the series. The *Gaujac* programs executes different paths depending upon the specified iteration of a loop. The *Sprsin* program does not perform a computation for a single column (the diagonal element) of each row of a matrix. The *Summinmax* program determines the minimum and maximum of each corresponding pair of elements in two vectors and these two tests are logically correlated. The first four programs in Table 4 can be found in the second edition of the *Numerical Recipes in C* text [12].

The results of evaluating these programs are shown in Table 5. For each program a direct-mapped instruction cache configuration containing 8 lines of 16 bytes was used. It was assumed that a cache hit required one cycle, a cache miss required ten cycles, and all data cache references were assumed to be hits. This is the same cache configuration that was used in previous timing analysis studies [9, 7, 5]. The *Observed Cycles* represent the cycles required for an execution with worst-case input data.[2] The number of cycles was measured by enhancing a traditional cache simulator [13] to perform pipeline simulation. The *Value Independent* and *Value Dependent Estimated Cycles* indicate the number of cycles estimated by the timing analyzer without and with using value-dependent constraints, respectively. An *Estimated Ratio* is the *Estimated Cycles* divided by the *Observed Cycles*.

The results show that exploiting value-dependent constraint information in a timing analyzer can significantly tighten WCET predictions. The programs *Frenel* and *Sumoddeven* execute alternating paths in a loop depending upon a flag variable. One of the alternating paths has a slightly longer WCET than the other path in both of these

[2] We modified the desired relative error of the *Expint* and *Gaujac* programs so they would not converge early, which allowed us to obtain an accurate maximum iterations for a loop and worst-case input data for the *Observed Cycles* in Table 5.

Name	Description or Emphasis
Expint	Computes an exponential integral.
Frenel	Computes noncomplex Fresnel integrals.
Gaujac	Computes the abscissas and weights of a 10 point Gauss-Jacobi quadrature formula.
Sprsin	Converts a 20x20 integer matrix into row-indexed sparse storage mode.
Summidall	Sums the middle half and all elements of a 1,000 integer vector.
Summinmax	Sums the minimum and maximum of the corresponding elements of two 1,000 integer vectors.
Sumnegpos	Sums the negative, positive, and all elements of a 1,000 integer vector.
Sumoddeven	Sums the odd and even elements of a 1,000 integer vector.

Table 4: Test Programs That Are Constrained by Dependencies on Data Values

Name	WCET Timing Prediction Results					Seconds Required for Analysis		
	Observed	Value Independent		Value Dependent		Previous	Current	Time
	Cycles	Estimated Cycles	Estim. Ratio	Estim. Cycles	Estim. Ratio	Analysis Time	Analysis Time	Ratio
Expint	58,397	1,292,086	22.126	58,471	1.001	0.382	0.300	0.785
Frenel	47,749	48,887	1.029	47,783	1.001	0.322	0.272	0.845
Gaujac	786,386	797,116	1.014	794,334	1.010	2.737	1.845	0.674
Sprsin	28,339	28,608	1.009	28,404	1.002	0.107	0.113	1.056
Summidall	15,340	18,090	1.179	15,341	1.000	0.060	0.052	0.867
Summinmax	16,080	17,080	1.062	16,080	1.000	0.067	0.050	1.034
Sumnegpos	11,067	13,068	1.181	11,068	1.000	0.050	0.037	0.746
Sumoddeven	15,093	16,102	1.067	15,099	1.000	0.038	0.038	1.000
Average	122,306	278,880	3.708	123,323	1.002	0.470	0.338	0.876

Table 5: WCET Prediction and Analysis Overhead Results of the Test Programs

programs. The timing analyzer was able to determine that longer path of each program could only be executed for one half of the iterations, which reduced the overestimations. The *Summinmax* and *Sumnegpos* programs have logically correlated branches and the timing analyzer was able to detect for each program that the longest path was infeasible due to this correlation. The compiler detected iteration-based constraints for the *Gaujac* and *Summidall* programs indicating that certain paths could only be executed in specific iterations. There was little overestimation in the previous version of the timing analyzer for *Gaujac* since these iteration-based constraints were associated with paths that were not in the most deeply nested loop of the program. However, *Summidall*'s iteration-based constraints were for the most frequently executed portion of that program and a significant overestimation of WCET was avoided. Finally, the compiler detected an iteration-based constraint in *Sprsin* and *Expint* that was associated with an equality test between a loop variable and a value that was invariant for that loop. This means that the loop could only execute a path associated with the equality transition from the block containing the test for a single iteration of the loop. For *Sprsin* this path required a smaller WCET than when the loop variable was not equal to the loop-invariant value. Thus, the overestimation by the previous version of the analyzer was quite small and would decrease when applied to arrays with larger dimensions. However, the opposite situation occurs in *Expint*, which has a higher WCET associated with the path where the loop variable is equal to the loop-invariant value. Thus, exploiting this value-dependent constraint significantly reduces the WCET overestimation of *Expint*.

The slight remaining overestimations for several of the programs in the current version of the timing analyzer were due to two reasons. First, occasionally conservatively categorized instructions hit in cache due to the order paths were executed because of dependencies on data values. Inaccuracies also resulted from instruction caching categorizations that change between loop levels and their interaction with the pipeline analysis [7].

Table 5 also shows execution time in seconds required to make predictions for the test programs for the previous (value-independent) and current (value-dependent) versions of the timing analyzer. The times were obtained by calculating for each program the average of the elapsed times of ten executions of the timing analyzer on a Ultra-SPARC. The decrease in elapsed time for the analysis was due to two reasons. First, we modified the timing analyzer to avoid redundant analysis of a path when its caching behavior has not changed. Second, the new approach does not analyze a path in a given iteration when the path was infeasible, its maximum iterations had been exhausted, or only required iterations of other paths were available.

6. Future Work

There are additional aspects of using value-dependent constraints in timing analysis that can be investigated. First, we intend to enhance the timing analyzer to use value-dependent constraints when estimating best-case execution times (BCETs). In fact, we believe that value dependency analysis will significantly tighten BCET predictions since these predictions often suffer from large underestimations due to dependencies on data values. For instance, the BCET prediction for a program like *Sprsin* that only performs calculations for the off-diagonal elements of a matrix (when nested loop variables have different values) suffers from a significant underestimation since there is only one diagonal element for each row. Second, many value-dependent constraints were not detected due to function calls separating effects and the branches affected. These value-dependent constraints could be detected using interprocedural analysis.

7. Conclusions

This paper has described how value-dependent constraints were automatically detected by a compiler and exploited by a timing analyzer. We described techniques to efficiently detect constraints from effects causing the outcome of a branch to become known and ranges of iterations associated with branch outcomes. This constraint information could be used by a variety of timing analyzers, including those that use an ILP solver. We presented algorithms that show how value-dependent constraints were used in a non-ILP based timing analyzer to constrain the minimum and maximum iterations associated with each path in a loop and how these path constraints were used in WCET loop analysis. Finally, we showed results from a number of test programs whose worst-case paths were constrained by dependencies on data values. These results indicate that detection and exploitation of value-dependent constraints can significantly tighten WCET timing predictions. Furthermore, the approaches used for detection and exploitation of value-dependent constraints were shown to be quite efficient and are fully automated, requiring no interaction from the user.

8. Acknowledgements

The authors thank Jack Davidson for allowing *vpo* to be used for this research. John Mellor-Crummey, Andreas Ermedahl, Mikael Sjödin, and the anonymous reviewers provided several helpful suggestions that improved the quality of the paper. This work was supported in part by NSF grant EIA-9806525.

9. References

[1] 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*, pp. 298-307 (December 1995).

[2] G. Ottosson and M. Sjödin, “Worst Case Execution Time Analysis for Modern Hardware Architectures,” *ACM SIGPLAN Workshop on Language, Compiler, and Tools for Real-Time Systems*, pp. 47-55 (June 1997).

[3] 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).

[4] T. Lundqvist and P. Stenström, “Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques,” *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 1-15 (June 1998).

[5] C. A. Healy, M. Sjodin, V. Rustagi, and D. B. Whalley, “Bounding Loop Iterations for Timing Analysis,” *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 12-21 (June 1998).

[6] F. Mueller and D. B. Whalley, “Avoiding Conditional Branches by Code Replication,” *Proceedings of the SIGPLAN ’95 Conference on Programming Language Design and Implementation*, pp. 56-66 (June 1995).

[7] 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).

[8] 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).

[9] 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).

[10] 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).

[11] Nagham M. Al-Yaqoubi, *Reducing Timing Analysis Complexity by Partitioning Control Flow,* Masters Project, Florida State University, Tallahassee, FL (1997).

[12] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing, Second Edition,* Cambridge University Press, New York, NY (1996).

[13] J. W. Davidson and D. B. Whalley, “A Design Environment for Addressing Architecture and Compiler Interactions,” *Microprocessors and Microsystems* **15**(9) pp. 459-472 (November 1991).