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 Öttoisson 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

\[
\begin{align*}
\%g1 &= \text{sethi} \%g1, 1; \\
\%o0 &= \text{cmp} \%o0, 5; \\
\text{PC} &= \text{IC} < 0, \text{L20}; \\
\text{IC} &= \text{IC} < 0, \text{L20};
\end{align*}
\]

Expanded Comparison

\[
\text{IC} = \text{R}[\%g1, \%o0];
\]

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 \( O(B^2C) \) 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 \( i \).

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 another conditional branch may indicate the direction taken by another conditional branch. If block 2 \( a[i] \geq 0 \) falls into block 3, then the value of \( a[i] \) is negative and block 5 \( a[i] \leq 0 \) 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].

![Figure 3: Logical Correlation between Branches](image)

### 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 0.099 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 \((== \text{ or } !=)\), 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.

![Figure 4: Ranges of Iterations and Branch Outcomes](image)
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](image)

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

<table>
<thead>
<tr>
<th>Loop</th>
<th>Total Iters</th>
<th>Path ID</th>
<th>Exit Path</th>
<th>Continue Path</th>
<th>Possible Iterations</th>
<th>Unique Iterations</th>
<th>Minimum Iterations</th>
<th>Maximum Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>1,001</td>
<td>1</td>
<td>Y</td>
<td>N</td>
<td>[1001..1000]</td>
<td>∅</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Figure 2</td>
<td>1,000</td>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>[1..1000]</td>
<td>∅</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Figure 3</td>
<td>1,000</td>
<td>1</td>
<td>Y</td>
<td>Y</td>
<td>[751..1000]</td>
<td>∅</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

-5-
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.

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].

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].
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 a conservative timing prediction. Fortunately, inequality tests (<, <=, >=, >) on loop induction variables do not occur frequently.

<table>
<thead>
<tr>
<th>Path</th>
<th>Possible Iterations</th>
<th>Min Iters</th>
<th>Max Iters</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1..50]</td>
<td>0</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>[1..50]</td>
<td>0</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>[51..100]</td>
<td>0</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>[51..100]</td>
<td>0</td>
<td>50</td>
<td>2</td>
</tr>
</tbody>
</table>

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 $2P$ 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 */
  IF req_iters < N - iters_handled THEN
    Find longest path P where
    P->req_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 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 mini-
imum 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 itera-
tions remaining in the loop.

5. Results

Table 4 depicts programs where the worst-case paths
were constrained by dependencies on data values to eval-
uate the effectiveness of detecting and exploiting value-
dependent constraints. The Sumoddeven, Sumnegpos, and
Summidall programs correspond to the examples illus-
trated in Figures 2, 3, and 4, respectively. The Expint pro-
gram 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 pro-
grams executes different paths depending upon the speci-
fied iteration of a loop. The Sprosis program does not per-
form a computation for a single column (the diagonal ele-
ment) of each row of a matrix. The Summinmax program
determines the minimum and maximum of each corre-
sponding 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 Numeri-
cal 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 refer-
ces 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. The number of cycles was measured by enhancing a
traditional cache simulator [13] to perform pipeline simu-
lation. The Value Independent and Value Dependent Esti-
mated 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 con-
straint 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.
Table 4: Test Programs That Are Constrained by Dependencies on Data Values

<table>
<thead>
<tr>
<th>Name</th>
<th>WCET Timing Prediction Results</th>
<th>Seconds Required for Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Observed Cycles</td>
<td>Value Independent</td>
</tr>
<tr>
<td></td>
<td>Estimated Cycles</td>
<td>Estim. Ratio</td>
</tr>
<tr>
<td>Expint</td>
<td>58,397</td>
<td>1,292,086 22.126</td>
</tr>
<tr>
<td>Frenel</td>
<td>47,749</td>
<td>48,887 1.029</td>
</tr>
<tr>
<td>Gaujac</td>
<td>786,386</td>
<td>797,116 1.014</td>
</tr>
<tr>
<td>Sprsin</td>
<td>28,339</td>
<td>28,608 1.009</td>
</tr>
<tr>
<td>Summidall</td>
<td>15,340</td>
<td>18,090 1.179</td>
</tr>
<tr>
<td>Summinmax</td>
<td>16,080</td>
<td>17,080 1.062</td>
</tr>
<tr>
<td>Sumnegpos</td>
<td>11,067</td>
<td>13,068 1.181</td>
</tr>
<tr>
<td>Sumoddeven</td>
<td>15,093</td>
<td>16,102 1.067</td>
</tr>
<tr>
<td>Average</td>
<td>122,306</td>
<td>278,880 3.708</td>
</tr>
</tbody>
</table>

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

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 Sprosin 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