Automatic Detection and Exploitation of Branch Constraints for Timing Analysis*

Christopher A. Healy
Computer Science Dept., Furman University
Greenville, SC 29613
e-mail: chris.healy@furman.edu, phone: (864) 294-2233

David B. Whalley
Computer Science Dept., Florida State Univ.
Tallahassee, FL 32306-4530
e-mail: whalley@cs.fsu.edu, phone: (850) 644-3506

Abstract

Predicting the worst-case execution time (WCET) and best-case execution time (BCET) 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 and underestimations of BCET 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 branch 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 branch constraints by a compiler and automatically exploiting these constraints within a timing analyzer. The result is significantly tighter timing analysis predictions without requiring additional interaction with a user.

Index terms: real-time systems, worst-case execution time, best-case execution time, timing analysis, infeasible paths, branch constraints

1. Introduction

Obtaining accurate worst-case execution time (WCET) and best-case execution time (BCET) predictions of programs is a challenging task. However, there is a significant amount of work in real-time scheduling that depends knowing the WCET of tasks in a system. Without tight WCET predictions, it could not be determined if many real-time systems would meet their timing requirements. One common practice is to estimate WCET and BCET bounds by measuring execution time with what a real-time programmer believes is WCET and BCET input data. Unfortunately, it is difficult to derive such input data given complex architectural features and/or complicated control flow in a program. This can lead a user to believe that timing requirements have been met, when in reality these requirements may actually be violated during critical situations. Performing a timing analysis automatically with a timing analysis tool is a much more desirable solution.

*This work was supported in part by the National Science Foundation under grant number EIA-9806525. A preliminary version of this work was described in the 1999 *Real-Time Technology and Applications Symposium* under the title "Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints."

Various features of the architecture, such as caches and pipelines, can affect the execution time of a sequence of instructions and these features need to be modeled while analyzing the control flow of a program. Even with perfect architectural modeling, significant overestimations of WCET and underestimations of BCET 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 in a program. We refer to such dependencies as branch constraints. While branch 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 contribution described in this paper includes techniques for automatically detecting branch constraints by a compiler and efficiently exploiting these constraints by a timing analyzer to obtain tighter timing predictions. The remainder of the paper has the following organization: First, we describe related work in this area. Second, we explain techniques to automatically detect branch constraints in a program. Third, we illustrate methods to convert the branch constraints into path constraints indicating how many times each path can be executed in a loop. Fourth, we depict how to use the path constraints in loop analysis to tighten the WCET and BCET predictions. Fifth, we give results for a number of applications that show that significant improvements in timing prediction accuracy can be obtained by automatically detecting and exploiting branch constraints. Finally, we describe future work in this area and give the conclusions for the paper.

2. Related Timing Analysis Work

Some constraint-based timing analyzers use branch constraints to obtain more accurate estimations of execution time. Li *et al*. performed timing analysis using an Implicit Path Enumeration Technique [1]. This technique used 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 branch constraints, which they called *program functionality constraints*. The work of Ottosson and Sjödin [2] extended the Implicit Path Enumeration Technique by using finite domain constraints to model the architectural features of the hardware. However, in both approaches these branch constraints had to be 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] uses symbolic execution to automatically resolve many branch constraints. The approach used by these authors is quite powerful, but effectively requires simulating all paths of a loop for every loop iteration. Thus, symbolic execution requires significant analysis overhead, which would be undesirable when analyzing long running programs.

Another type of branch constraint is the number of iterations associated with a loop. We have implemented techniques to automatically determine the minimum and maximum number of 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, 6]. The symbolic execution approaches [3, 4] also provide a more powerful and less efficient method to calculate bounds on the number of loop iterations. In this paper, we address detecting and exploiting branch constraints that constrain execution paths rather than the number of iterations that a loop can execute.

3. Automatic Detection of Branch Constraints

A branch constraint causes the outcome of a conditional branch to be known under certain conditions. The authors implemented techniques that commonly detect these conditions. These techniques include detecting effect-based constraints by analyzing the effect that an assignment to a variable will have on a branch and detecting that the outcome of one branch has a logical correlation with the outcome of another branch. In fact, we have used a similar type of analysis to detect branches that could be avoided by duplicating code [7]. In addition, we detect iteration-based constraints by using value range analysis to determine the frequency that a branch will fall through or be taken. This was accomplished by determining the iterations in which each path p may be executed. This value range analysis is similar to analysis used for compiler optimizations for obtaining predictions on the percentage of time that a branch will be taken or fall-through [8]. Value range analysis was also used to help determine the minimum and maximum number of iterations for loops in a program [5, 9].

3.1. Detecting Effect-Based Constraints

Analysis is performed in the compiler 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 conditional branch (and its associated comparison instruction) depends. This set is calculated by expanding the effects of the comparison instruction associated with the conditional branch. For instance, consider the SPARC instructions represented as RTLs (Register Transfer Lists) and the associated expanded comparison, shown 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 is encountered with multiple predecessors. 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 by instructions for each basic block. Each conditional 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.

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 in the function. Note that all of the branch constraint analysis presented in this paper was performed within a function (intra-procedural analysis, not inter-procedural analysis).

Consider the source code in Figure 2(a). The corresponding control flow that is generated by the compiler is shown in Figure 2(b), constraints are shown in Figure 2(c), and paths are shown in Figure 2(d). Paths will be discussed in Section 4. While the control flow Figure 2(b) 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

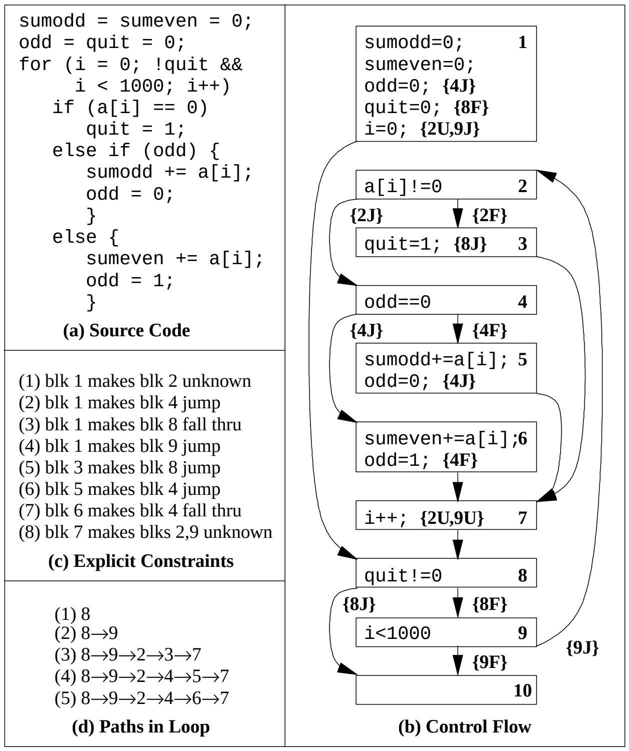
Instructions in a Basic Block

```
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[8]?5; => IC=R[r[1]+LO[_g]]?5; => IC=R[HI[_g]+LO[_g]]?5;
```

Figure 1: Example of Expanding a Comparison



```
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 makes blk 2 unknown
(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 makes blks 2,9 unknown

(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

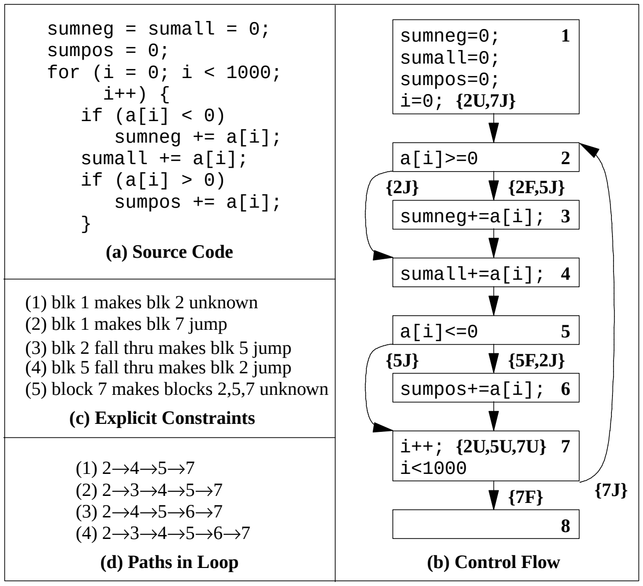
(b) Control Flow

Figure 2: Example Illustrating Effects of Assignments on Branches

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 branch 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 both depend on the value of `i`.

Figure 2(b) also shows implicit branch 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.

There are also situations where one conditional branch may be logically correlated with another conditional branch. In other words, the direction taken by one conditional branch may indicate the direction taken by another conditional branch. The source code in Figure 3(a) and corresponding control flow in Figure 3(b) depict such a situation. 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 branch 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



```
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 makes blk 2 unknown
(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 makes blocks 2,5,7 unknown

(c) Explicit Constraints

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

(d) Paths in Loop

(b) Control Flow

Figure 3: Example Illustrating a Logical Correlation between Branches

of conditional branches. Note that a branch is always logically correlated with itself and these self correlations are implicit constraints. The example of the paths shown in Figure 3(d) will be described in Section 4.

The exact conditions when it is known that one branch is logically correlated with another branch have been described in previous work [7] and are depicted in Table 1. In general, a conditional branch can only be correlated with another branch when one argument of each comparison is identical and the other argument of each comparison is a constant or the same invariant value. Table 1 depicts the different cases when the result of one branch is correlated with another branch. Column 1 shows a known result from one branch. This result is determined by not only the operands of the comparison and the branch relational operator, but also by whether or not the branch was taken. The second column in Table 1 depicts the condition associated with the correlated branch. The third and fifth columns of Table 1 define the requirements for the correlated branch to jump or fall through, respectively.

Table 1: Logically Correlated Branch Requirements

known result	correlated branch	jump requirement	example	fall through requirement	example
v = c1	v = c2	c1 = c2	v = 10 → v = 10 since 10 = 10	c1 ≠ c2	v = 10 → ¬(v = 15) since 10 ≠ 15
	v ≠ c2	c1 ≠ c2	v = 10 → v ≠ 15 since 10 ≠ 15	c1 = c2	v = 10 → ¬(v ≠ 10) since 10 = 10
	v rel2 c2	c1 rel2 c2	v = 10 → v < 20 since 10 < 20	¬(c1 rel2 c2)	v = 10 → ¬(v > 20) since ¬(10 > 20)
v ≠ c1	v = c2	N/A	N/A	c1 = c2	v ≠ 10 → ¬(v = 10) since 10 = 10
	v ≠ c2	c1 = c2	v ≠ 10 → v ≠ 10 since 10 = 10	N/A	N/A
v rel1 c1	v rel2 c2	addeq(rel1) = addeq(rel2) && c1* addeq(rel1) c2*	v ≥ 11 → v > 10 since ’≥’ = ’≥’ && 11 ≥ 10+1	opp(noeq(rel1), noeq(rel2)) && ¬(c1* addeq(rel2) c2*)	v ≥ 10 → ¬(v < 10) since opp(’>’, ’<’) && ¬(10 ≤ 10-1)
	v = c2	N/A	N/A	c1 noeq(rel1) c2	v ≥ 20 → ¬(v = 10) since 20 > 10
	v ≠ c2	c1 noeq(rel1) c2	v ≥ 20 → v ≠ 10 since 20 > 10	N/A	N/A

where
(1) v is a variable
(2) c is a constant
(3) rel is ’<’, ’≤’, ’>’, or ’≥’
(4) opp(rel1, rel2) returns true when (x rel1 y) && (x rel2 y) can never both be true (e.g. x > y && x < y)
(5) noeq(rel) returns the relational operator without any equality (e.g. noeq(’≥’) and noeq(’>’) both return ’>’)
(6) addeq(rel) returns the relational operator with an equality (e.g. addeq(’≥’) and addeq(’>’) both return ’≥’)
(7) c* is a constant that is adjusted by 1 in the appropriate direction if addeq(rel) != rel

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. For each of these branches, the compiler derives the information shown in Table 2.

If the branch meets all of the requirements in Table 2, then the compiler next calculates on which iteration the branch will change direction, which is determined using Equation 1. Table 3 depicts the various cases in which the branch condition will always or never be satisfied, and also how the compiler determines the appropriate value to use for *adjust* in Equation 1. Once the compiler has determined the value of *I*, it produces directives for a timing analyzer indicating ranges of iterations for each of the two outgoing edges from the block containing the branch. The *relop* and the direction of the increment (i.e. the sign of *before*+*after*) are used to determine which edge is taken on the first *I*–1 iterations.

$$I = \left\lfloor \frac{limit - (initial + before) + adjust}{before + after} \right\rfloor + 2 \qquad (1)$$

Table 2: Information Calculated for Each Iteration Branch

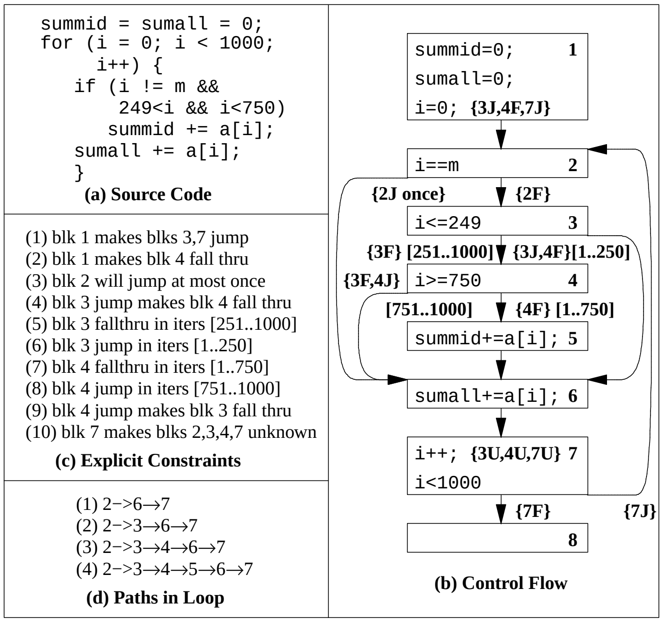
Term	Explanation	Requirement
variable	The control variable on which the branch depends, which is the variable or register being compared to a constant in the block containing the branch.	The control variable must be a basic induction variable, which is an integer variable `v` whose only assignments within the loop are of the form `v := v ± c` where `c` is an integer constant [10].
limit	The value being compared to the *variable* in the block containing the branch.	The limit must be an integer constant.
relop	The relational operator used to compare the *variable* and the *limit*.	
initial	The value of the *variable* when the loop is entered.	The initial value must be an integer constant.
before	The amount by which the *variable* is changed before reaching the branch in each iteration.	The amount by which the control variable is incremented or decremented must be an integer constant and these constant changes must occur on each complete iteration of the loop.
after	The amount by which the *variable* is changed after reaching the branch in each iteration.	The amount by which the control variable is incremented or decremented must be an integer constant and these constant changes must occur on each complete iteration of the loop.
adjust	An adjustment value of –1, 0, or 1, which compensates for the difference between relational operators (e.g. < and ≤).	

Table 3: How to Determine When a Branch Changes Direction

Operator	Condition	Test Result	*adjust*
<=	$\mathit{first} \le \mathit{limit}$ & $\mathit{incr} > 0$	is false on the Ith iteration	0
<=	$\mathit{first} \le \mathit{limit}$ & $\mathit{incr} \le 0$	always true	
<=	$\mathit{first} > \mathit{limit}$ & $\mathit{incr} \ge 0$	always false	
<=	$\mathit{first} > \mathit{limit}$ & $\mathit{incr} < 0$	is true on the Ith iteration	1
<	$\mathit{first} < \mathit{limit}$ & $\mathit{incr} > 0$	is false on the Ith iteration	−1
<	$\mathit{first} < \mathit{limit}$ & $\mathit{incr} \le 0$	always true	
<	$\mathit{first} \ge \mathit{limit}$ & $\mathit{incr} \ge 0$	always false	
<	$\mathit{first} \ge \mathit{limit}$ & $\mathit{incr} < 0$	is true on the Ith iteration	0
>	$\mathit{first} \le \mathit{limit}$ & $\mathit{incr} > 0$	is true on the Ith iteration	0
>	$\mathit{first} \le \mathit{limit}$ & $\mathit{incr} \le 0$	always false	
>	$\mathit{first} > \mathit{limit}$ & $\mathit{incr} \ge 0$	always true	
>	$\mathit{first} > \mathit{limit}$ & $\mathit{incr} < 0$	is true on the Ith iteration	1
>=	$\mathit{first} < \mathit{limit}$ & $\mathit{incr} > 0$	is true on the Ith iteration	−1
>=	$\mathit{first} < \mathit{limit}$ & $\mathit{incr} \le 0$	always false	
>=	$\mathit{first} \ge \mathit{limit}$ & $\mathit{incr} \ge 0$	always true	
>=	$\mathit{first} \ge \mathit{limit}$ & $\mathit{incr} < 0$	is false on the Ith iteration	0

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

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 by using Equation 1. 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 non-constant 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 paths shown in Figure 4(d) will be described in Section 4. The detection of iteration-based constraints requires $O(C)$ complexity, where C is the number of conditional branches, since each branch must be inspected once. Note that this detection of iteration-based constraints takes place after the compiler has performed induction variable analysis.



```
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 makes blks 2,3,4,7 unknown

(c) Explicit Constraints

(1) 2−>6→7
(2) 2−>3→6→7
(3) 2−>3→4→6→7
(4) 2−>3→4→5→6→7

(d) Paths in Loop

(b) Control Flow

Figure 4: Example Illustrating Ranges of Iterations Associated with Branch Outcomes

4. Exploiting Branch Constraints in a Timing Analyzer

The analysis techniques described in the previous section to identify branch 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 effectively modeled across multiple blocks. More importantly, the time required for the analysis does not scale well with an ILP approach since thousands of constraints may have to be solved for even relatively small programs. Some programs that required only a few seconds of timing analysis using more traditional approaches [11, 12] 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 scalability problems. Finally, when a timing requirement is violated, a user would like to know where the time is being spent in the code associated with the constraints. The timing analysis approach described in this paper not only produces WCET and BCET predictions for an entire program, but also gives the WCET and BCET for each function, loop and path in the program [13]. In contrast, an ILP approach only calculates a single WCET and BCET prediction for the entire program. Thus, the authors decided it would be worthwhile to investigate how branch constraints could be exploited by a non-ILP based timing analyzer.

Figure 5 depicts the overall organization of the non-ILP timing analysis environment that was modified to exploit branch constraint information. An optimizing compiler [14] was used to produce control flow and branch constraint information as a side effect of the compilation of a file. This information includes the number of iterations associated with loops in the program.[1] [5] A static instruction 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 [15]. A separate categorization is given for each loop level in which the instruction is contained. These categorizations are described in Table 4. Data caching categorization and analysis was not used in this study. 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 [16, 11, 17].

Given a program's control-flow information and instruction caching categorizations along with the processor's instruction set information, the timing analyzer then derives best-case and worst-case estimates for each path, loop

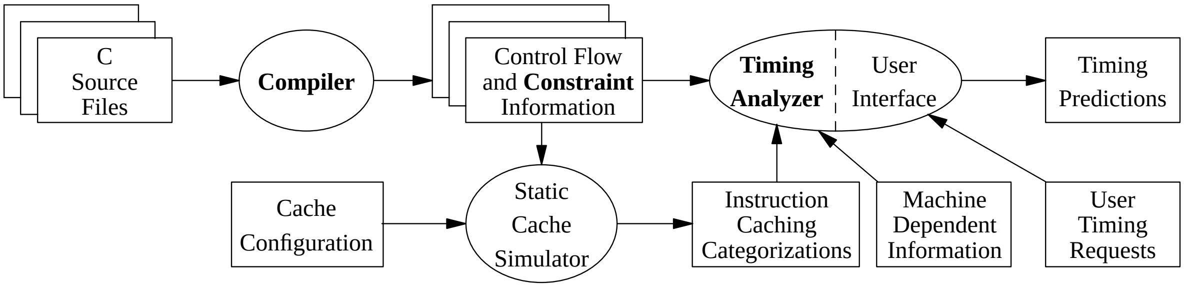


Figure 5: Overview of Process to Obtain Timing Predictions

[1] If the number of iterations cannot be determined by compile-time analysis, then the user is prompted for the minimum and maximum values of the variables on which the loop depends. Likewise, the user can specify this information as assertions within the source code [5, 9]. Specifying minimum and maximum values of variables is much safer than specifying the number of loop iterations since the user is not always aware of the code generation strategies or optimizations performed by the compiler that may affect the number of iterations executed in a loop.

Table 4: Definitions of Instruction Categories for Worst-Case Analysis

Caching Category	Definition
always miss	The instruction is not guaranteed to be in cache when it is referenced.
always hit	The instruction is guaranteed to be in cache when it is referenced.
first miss	The instruction is not guaranteed to be in cache on its first reference each time the loop is executed, but is guaranteed to be in cache on subsequent references.
first hit	The instruction is guaranteed to be in cache on its first reference each time the loop is executed, but is not guaranteed to be in cache on subsequent references.

and function within the program. To statically estimate the caching behavior of a program as accurately as possible, functions are distinguished by function instances. An instance depends on the calling sequence, that is, it depends on the immediate call site within its caller as well as the caller's call site, etc. The instance *i* of a function corresponds to the *i*th occurrence of the function within a depth-first traversal of the call graph [16]. A timing analysis tree is constructed, where each node of the tree corresponds to a loop or function in the function instance graph. Each node is considered a natural loop.[2] A node that represents a function instance is treated as a loop that will iterate exactly once when entered. The timing analyzer determines the set of possible paths for each node. The loops in the timing analysis tree are processed in a bottom-up manner. In other words, the WCET and BCET for a loop are not calculated until the times for all of its immediate child loops are known. This means that the timing analyzer determines execution time for programs by first analyzing the innermost loops and functions, and proceeding to higher level loops and functions until it reaches `main()`. After processing the `main` function, a graphical user interface is invoked that allows the user to request predictions for specified portions of the program [18].

The remainder of this section will describe the details of how the timing analyzer makes use of the branch constraints to compute the WCET and BCET predictions for a particular loop or function. In particular, constraints on paths are generated from the branch constraints. For example, effect-based branch constraints can be used to determine if a given path is infeasible, or that one path cannot follow some other path on a subsequent iteration of

[2] A natural loop is a loop with a single entry block. While the static simulator can process unnatural loops, the timing analyzer is restricted to only analyzing natural loops since it would be difficult for both the timing analyzer and user to determine the set of possible blocks associated with a single iteration in an unnatural loop. Likewise, the timing analyzer is also restricted to direct calls and nonrecursive programs. It is often difficult to automatically determine the set of functions that could be invoked with an indirect call. While cycles can be detected in a call graph and could be viewed as a loop, it would be difficult for a timing analyzer or a user to determine the number of iterations through such a cycle. It should be noted that unnatural loops and indirect calls occur quite infrequently in typical C applications.

the loop. Further constraints arise from analyzing which paths can execute on the first iteration. For each path *p*, iteration-based constraints are used to determine the range of iterations in which *p* may execute. Once the path constraints have been calculated, they are used in the worst-case and best-case loop analysis algorithms. The purpose of using these path constraints is to tighten the execution time predictions. For instance, if the timing analyzer can determine that the longest (shortest) path is infeasible or can only execute for a proper subset of the loop's iterations, then the WCET (BCET) bound will be tighter.

4.1. Analyzing Branch Constraints to Create Path Constraints

The timing analyzer uses the branch constraints to calculate a minimum and maximum number of iterations associated with each path during the execution of a loop. Table 5 depicts worst-case information associated with each loop path described in Figures 2(d), 3(d), and 4(d). Table 6 shows the analogous best-case path iteration information for each loop path described in Figure 2(d). The second and third example loops are not shown in Table 6 because their best case iteration information is identical to their worst case information from Table 5. The first loop example from Figure 2 does have a different number of iterations for worst case and best case, and this results in a different set of possible iterations and number of maximum iterations for each path. The total number of loop iterations is automatically calculated using techniques described in previous work [5]. A *loop 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. *Possible iterations* indicate in which iterations the specified path can possibly be taken. *Unique iterations* indicate in which iterations only the specified path could be taken. The possible and unique iterations are used to constrain the maximum and minimum number of iterations in which a path can be taken, which are shown in the final two columns. If the timing analyzer determines that a path *p* may be taken on at most one iteration, then *p* is called a *once* path. The presence of a once path in a loop causes the unique range and the minimum number of iterations corresponding to each of the other paths to be reduced by one. For instance, for the loop in Figure 4, path 1 is a once path. Consequently, the unique range and minimum iteration information for paths 2-4 are updated to reflect the possibility that path 1 may execute one time.

Table 5: Worst-Case Path Information for Figures 2(d), 3(d), and 4(d)

Example	Total Loop Iterations	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]	[1..1]	1	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 6: Best-Case Path Information for Figure 2(d)

Loop	Total Iters	Path Type	Path ID	Possible Iterations	Unique Iters	Min Iters	Max Iters
Loop in Figure 2	2	exit	1	[2..2]	∅	0	1
		exit	2	[2..2]	∅	0	1
		cont	3	[1..1]	∅	0	1
		cont	4	N/A	∅	0	0
		cont	5	[1..1]	∅	0	1

Figure 6 gives a high-level description of the algorithm used to calculate the information given in the last four columns of Table 5. The algorithm is organized into eleven steps. Except for the construction of the REACH_SELF table in step 3, the complexity of the algorithm is $O(P^2)$, where P is the number of paths in the loop.[3] 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. The following section provides examples to illustrate how this information is calculated.

4.2. Using Effect-Based Constraints

Effect-based constraints are either 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.

[3] If the number of paths within a loop exceeds a reasonable limit, then the control flow is partitioned to reduce the timing analysis complexity [19].

```
struct path {
   struct range_node range;              /* iterations when path can be taken                                        */
   struct range_node uniqrange;          /* iterations when only this path can be taken                              */
   boolean once;                         /* path can be taken at most once for each loop execution                   */
   int nonuniqiters;                     /* number of iterations when other overlapping paths can be taken at most once */
   int miniters;                         /* minimum times path can be taken for a loop execution                     */
   int maxiters;                         /* maximum times path can be taken for a loop execution                     */
   int set;                              /* set of paths in loop with overlapping ranges with this path              */
};

boolean CAN_FOLLOW[numpaths][numpaths];  /* can one path follow another                                              */
int REACH_SELF[numpaths];                /* number of iterations before a path can follow itself                     */
```

```
/* 1. disregard 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
      Disregard P from the analysis.

/* 2. 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.

/* 3. 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.

/* 4. 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.

/* 5. 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..max(N-1,1)].
   IF P is an exit path THEN
      P->range = P->range ∪ [N..N].
```

```
/* 6. 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
      Disregard P from the analysis.

/* 7. 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 non-leaf 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.

/* 8. 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 DO
      P->uniqrange = P->uniqrange − Q->range.

/* 9. 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.

/* 10. assign maximum number of iterations for each path */
FOR each path P in the loop DO
   IF REACH_SELF[P] = 0 OR 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]).

/* 11. 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 5

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 disregards the path from the analysis to prevent any additional analysis time to be spent on it.

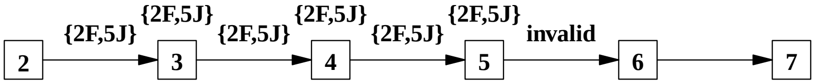


Figure 7: Path 4 in Figure 3(d) Is Not Feasible

The maximum number of iterations for a path can sometimes be constrained by information associated with 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. Branch 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 constraints for branch 4 reaching the end of paths 4 and 5 from Figure 2 contains the opposite outcome of branch 4 in their respective paths. This causes these paths not to be taken on the next loop iteration.

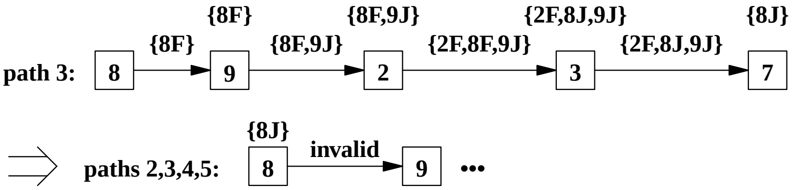


Figure 8: Paths 2, 3, 4, and 5 Cannot Follow Path 3 in Figure 2(d)

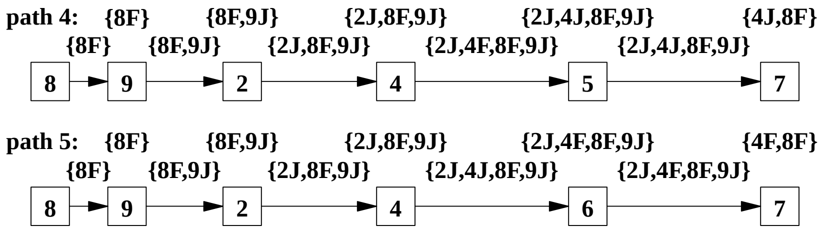


Figure 9: Paths 4 and 5 Cannot Immediately Follow the Same Path in Figure 2(d)

A CAN_FOLLOW matrix is constructed by the timing analyzer that indicates for each path the set of other paths that can legally follow it on the next iteration. If the 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 7 depicts the matrix for which paths can legally follow each path that are shown in Figure 2(d).

Table 7: Paths That Can Immediately Follow the Loop Paths in Figure 2

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

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 number of *K* paths 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 *M* shown in Equation 2, 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.

$$M = \left\lceil \frac{R}{K} \right\rceil \tag{2}$$

4.3. Using Effect-Based Constraints On Entering a Loop

The previous section discussed how branch constraints are used to create path constraints within a loop. But there are further constraints that arise when the loop is entered that affect when paths can initially execute. The steps taken by the timing analyzer related to these initial constraints are as follows.

1. use data-flow analysis to determine the initial constraints
2. determine the first iteration on which each path in the loop can execute
3. update the range of possible iterations for the paths
4. update the minimum and maximum number of iterations of the loop

These steps are described in this section.

The timing analyzer performs data-flow analysis [10] to calculate *ins* and *outs* for each block in a function. The algorithm for accomplishing this uses a standard data-flow technique, and is given in Figure 10. The inner FOR-loop in Figure 10 combines the effects from each predecessor block one at a time. The implementation uses

```
FOR each function in the program DO
   DO
      change = FALSE
      FOR each block in the function DO
         in.j = NULL
         in.f = NULL
         in.u = NULL
         IF the block has at least one predecessor (pred) THEN
            in.j = pred.out.j
            in.f = pred.out.f
            FOR each other predecessor block (pred) DO
               in.j ∩= pred.out.j
               in.f ∩= pred.out.f
            in.u = ~(in.j ∪ in.f)

         Initialize this.f, this.u and this.j based on the branch
            constraints contained in this block.
         out.j = this.j ∪ (in.j - this.f - this.u)
         out.f = this.f ∪ (in.f - this.j - this.u)
         out.u = this.u ∪ (in.u - this.j - this.f)
         IF any in or out bit vector changed THEN
            change = TRUE
   WHILE change
```

Figure 10: Calculating Ins and Outs

six bit vectors for each block: in.jump, in.fallthru, in.unknown, out.jump, out.fallthru and out.unknown. The jump, fallthru and unknown bit vectors indicate which branches are made to jump, fall through or become unknown, respectively, based on this block. For determining the ins and outs of a block, exactly one of the three corresponding bit vectors must be set, since a branch must be in either a jump, fall through or unknown state. Each block also contains bit vectors indicating if it causes a branch to jump, fall through or become unknown. However, the current block may have no effect on the branch in question, so it is possible that the bit vectors representing the effect from the current block may all be zero.

The data-flow equations 3 through 8 determine the ins and outs. Equations 3 and 4 show that the current block's ins for the jump (fall through) branches are simply the intersection of the jump (fall through) bit vectors of the predecessors' outs. Equation 5 states that the ins for the unknown branches are the complement of the union of the ins for the jump and fall through branches. For example, if one predecessor out says that a certain branch will fall through, but another predecessor out says the same branch will jump, then the in of the current block will show that that branch is unknown due to the conflict between the predecessors.

$$B.in.j = \bigcap_{p \in preds(B)} p.out.j \tag{3}$$

$$B.in.f = \bigcap_{p \in preds(B)} p.out.f \tag{4}$$

$$B.in.u = \overline{(B.in.j \cup B.in.f)} \tag{5}$$

$$B.out.j = B.j \cup (B.in.j - B.f - B.u) \tag{6}$$

$$B.out.f = B.f \cup (B.in.f - B.j - B.u) \tag{7}$$

$$B.out.u = B.u \cup (B.in.u - B.j - B.f) \tag{8}$$

Calculating the outs using Equations 6-8 is also straightforward. If the current block has no effect on a branch, then the out bit vectors will be assigned the value of the ins. Otherwise, the effect of this block will override the ins to determine the outs of this block. For example, consider a situation where the block in question makes a particular branch in block *i* jump, while the effect of the ins is to make that jump fall through. In other words, the value of bit position $B.j_i$ is 1, and the bit positions $B.in.j_i$, $B.f_i$ and $B.u_i$ equal 0, so that according to Equation 6 the value of $B.out.j_i$ is set to 1. In this case, since this block has an effect, it overrides the ins, so the value of the out bit vectors will represent that the branch will jump. Note that a block can have at most one effect on a given branch.

After the ins and outs of every block are calculated, the timing analyzer uses the outs of the preheaders to see which paths can execute on the first iteration. The algorithm in Figure 11 sets `p.on_first` to true (false) if it determines path `p` can (cannot) execute on the first iteration. The cases in which `p.on_first` is false correspond to situations where a branch in the path contradicts the information from the preheader outs. If a path is found not able to execute on the first iteration as a result of this algorithm, then in some cases it may be assigned fewer maximum iterations, and a more accurate timing bound can be obtained. The preheaders' *out* constraints are propagated through each path. Any path that does not obey the preheaders' constraints cannot execute on the first iteration. For

```
Initialize pre.j and pre.f to be the intersection of the
   respective bit vectors of all the header's immediate
   predecessors that are not in the loop.
Initialize pre.u to be the complement of the union
   of pre.j and pre.f.
FOR each path (p) in the loop DO
   IF we already know the path cannot execute on
      first iteration THEN
      CONTINUE
   p.on_first = TRUE
   FOR each block (b) in path p DO
      IF there is no branch in this block THEN
         CONTINUE
      IF all three bit vectors at bit b are zero THEN
         CONTINUE

      /* if the preheader says this branch must jump */
      pre.j[b] THEN
         IF this is not the last block in path THEN
            IF number of next block in path == b + 1
               p.on_first = FALSE
         ELSE
            IF a branch transition is part of a
               different path in the loop THEN
               p.on_first = FALSE

      /* if the preheader says this branch must fall through */
      ELSIF pre.f[b] THEN
         IF this is not last block in path THEN
            IF number of next block in path != b + 1
               p.on_first = FALSE
         ELSE
            IF a fall-through transition is part of a
               different path in the loop THEN
               p.on_first = FALSE
```

Figure 11: Which Paths Can Execute on First Iteration

example, consider the loop in Figure 2. The application of the algorithm in Figure 11 to the paths of this loop is depicted in Figure 12. This figure shows the propagation of the preheader constraints to determine which paths can execute on the first iteration. The solid arrows indicate transitions that occur between blocks inside the loop, while dashed arrows indicate transitions to or from a block outside the loop. Block 1 is the preheader of the loop, and block 10 is the block to which the loop exits. The value of `odd` is initialized to 0 in block 1, which is in the outs of the preheader of the loop, so the associated branch constraint is **{4J}**. Thus, on the first iteration of the loop, the branch in block 4 must be taken. Path 4 contains a transition from block 4 to block 5, which is a fall through situation, contradicting the preheader constraint. The timing analyzer detects that path 4 cannot execute on the first iteration.

The algorithm in Figure 11 also detects if a loop exit transition in a path causes it to be ineligible to execute on the first iteration. Consider exit paths 1 and 2 from the loop in Figure 2. Path 1 consists only of block 8, so this

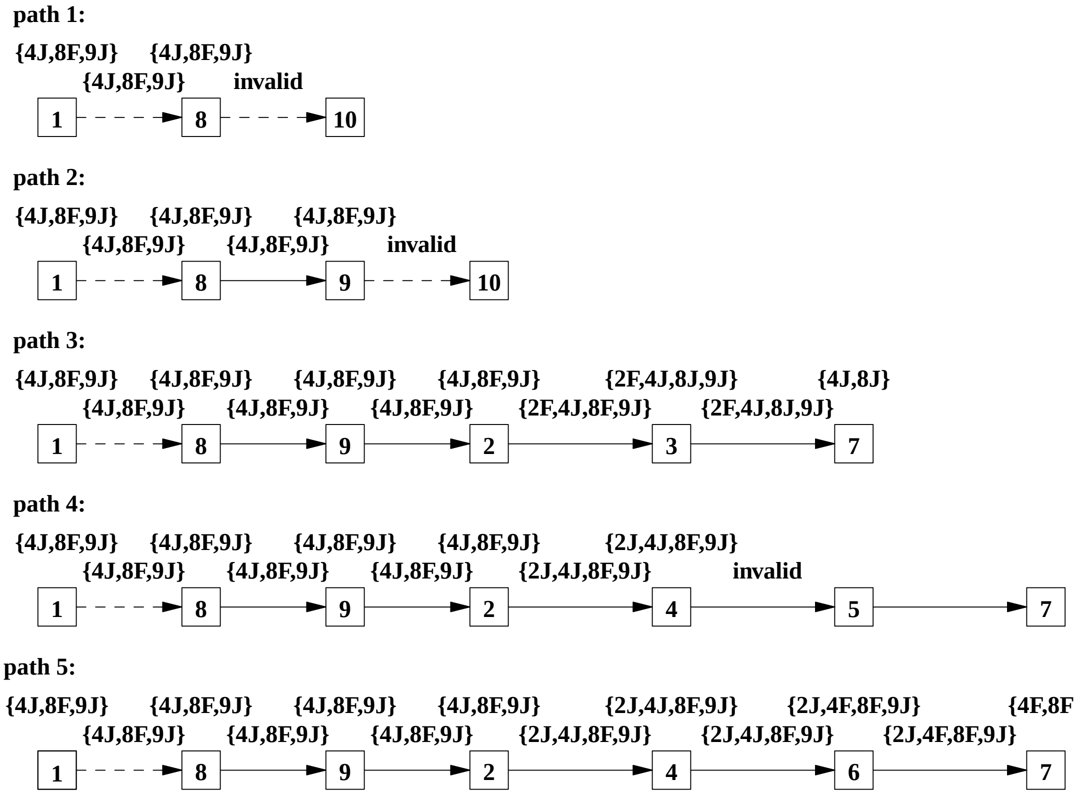


Figure 12: Propagating Preheader Constraints for Figure 2

block is considered the last block in the path. The timing analyzer determines the successor block to block 8 that is located outside the loop, which is block 10. The exit transition from block 8 to block 10 is a jump, however the preheader constraint is for the branch in block 8 to fall through (see **8F** constraint shown for path 1 in Figure 12). This contradiction means that path 1 cannot execute on the first iteration. Path 2 has a similar situation. Its last block is block 9, and its successor that is located outside the loop is block 10. To exit the loop by taking path 2 implies that the branch in block 9 must fall through, but the preheader constraint says that it must jump (see **9J** constraint shown for path 2 in Figure 12). So the timing analyzer concludes that path 2 cannot execute on the first iteration as well. For those paths that cannot execute on the first iteration, the next step is to determine on which iteration it can first be taken. Table 8 shows a Path Distance matrix for the example loop in Figure 2 that is derived from the Can Follow matrix given in Table 7. The table entries containing ∞ indicate that it is impossible for one path to reach the other path. For paths that cannot execute on the first iteration, the timing analyzer determines on which iteration it can execute as follows. Let *P* be the set of paths that can execute on the first iteration, and let *Q* be the set of paths that cannot. For each path *q* in *Q*, the timing analyzer finds the shortest number of iterations to reach *q* from any path in *P*. This shortest distance plus 1 represents the first iteration on which path *q* can execute. Continuing with the example from Figure 2, path 4 belongs to the set *Q*. Path 5 is a path in *P*, and according to Table 8 the path distance from path 5 to path 4 is one iteration. So the timing analyzer concludes that path 4 can first execute on the second iteration, and the range of possible iterations becomes [2..1000] in worst case and [2..2] for best case. However, in the best case, path 4 is not an exit path, but its range of possible iterations is [2..2], namely the last iteration. This is a contradiction since any path that can execute on the last iteration must be an exit path. Thus, path 4 is an infeasible

Table 8: Path Distance Matrix for Figure 2

Current Path in Loop	How Many Iterations to Reach Path				
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	1	∞	∞	∞	∞
4	2	1	1	2	1
5	2	1	1	1	2

path in best case, and its maximum iterations is set to 0 as shown in Table 6.

Similarly, the timing analyzer determined that exit paths 1 and 2 could not execute on the first iteration. However, the path distances from path 3 to path 1 and from path 5 to path 2 are both one iteration as indicated in Table 8. Since both path 3 and path 5 can execute on the first iteration, paths 1 and 2 can first execute on the second iteration of the loop. For best case analysis, their ranges of possible iterations are adjusted to [2..2] as shown in Table 6. Their worst-case possible iterations are not updated since they had already been determined to be [1001..1001] in Table 5.

The timing analyzer enforces a rule that if any exit path can execute on the first iteration, then it must allow all exit paths to be chosen for the first iteration. The reason for this rule is that in best case, the BCET is assumed to occur for the minimum number of iterations. Consider a loop having three paths, where paths A and C are exit paths and path B is a non-exit path. Path A can exit on the first iteration, but path C can only exit after executing path B. If path A is significantly longer than paths B and C, then it is possible that the execution of paths B and C (two iterations) may be shorter than the execution of path A (one iteration). The authors believe that requiring the best-case loop analysis algorithm to repeatedly examine a loop for varying numbers of iterations would be overly inefficient. Specifying the minimum number of iterations before starting loop analysis makes the algorithm much simpler and only slightly more conservative in this highly unlikely scenario. In the above scenario, the timing analyzer will make the conservative assumption that path C can execute on the first iteration, and that the minimum number of iterations is still one.

If it turns out that no exit path can execute on the first iteration, then the timing analyzer updates the number of iterations of the loop based on when the exit paths can execute. In the example from Figure 2, both exit paths can only execute on the second iteration, so the timing analyzer sets the minimum number of iterations to 2, even though the compiler had previously determined, before this path analysis was performed, that the minimum number of iterations would have been 1 [5].

4.4. Using Iteration-Based Constraints

The maximum number of iterations can sometimes be constrained by analyzing iteration-based constraints. Tables 3 and 4 also show the range of possible iterations that is associated with each path. 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 13 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].

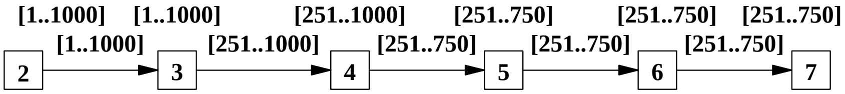


Figure 13: Iteration-Based Constraints Propagated Through Path 4 in Figure 4

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 7 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. This determines the unique set of iterations for the current path, which is the minimum number of times that the path has to be executed. 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), which is described in Section 3.2. 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.

4.5. Using the Path Constraints in Loop Analysis

The authors decided to use the minimum and maximum iterations associated with each loop path to obtain tighter loop predictions without restricting the order in which these paths are evaluated. There were several reasons for using this approach. First, the approach supports paths that can execute at most once, but in any iteration. For example, consider path 1 of the loop in Figure 4. This situation may occur frequently in numerical applications. Special conditions are often checked for the diagonal elements of a matrix (diagonal systems). Second, the approach deals with paths that have dependencies on other paths, such as paths 4 and 5 in Figure 2. Finally, the timing analyzer often calculates an average WCET and BCET 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, 9]. Using this approach allows the calculation of a safe average WCET (BCET) since the longest (shortest) paths are selected first in the respective loop analysis algorithms.

In addition, the timing analyzer determines 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. The timing analyzer uses 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 9 depicts an example with 4 paths and 2 sets. Each set of paths can only execute a maximum of 50 iterations. Consider only using the maximum iterations of each path , as opposed to checking if the set to which a path belongs has exhausted its number of iterations. 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. This 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. The two

Table 9: Example Illustrating Use of Path Sets

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

subsections that follow describe the worst-case and best-case loop analysis algorithms that employ the path constraint information.

Figure 14 shows how the WCET loop analysis algorithm uses the path constraint information. Let N be the maximum number of iterations and P be the number of paths in a loop. The DO-WHILE loop in step 3 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.  calculate required and non-required 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.

/* 2.  process all iterations of the loop */
iters_handled = 0.
pipeline_info = NULL.
WHILE iters_handled < N DO

   /* 3.  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

   /* 4.  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 14: WCET Loop Analysis Algorithm

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 is used in the loop analysis and correctness arguments about selecting the longest path using these categorizations and information have been given in previous work [16, 11]. Thus, it remains to be shown that each time a path is selected, it is really chosen from the set of paths that can still possibly execute given that the minimum and maximum number of iterations for each path was safely (but perhaps conservatively) estimated. A path's number of required iterations is its minimum iterations to be performed. The non-required 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 step 3 of Figure 14. If the iterations remaining is greater than the required iterations left to be 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 required iterations left to be performed. Step 4 of 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 non-required iterations of the path or of the entire loop. Therefore, the set of paths that can still possibly execute is accurate since a given path's required iterations are always processed before its non-required iterations and the number of non-required iterations to be processed for a path is never allowed to exceed the number of non-required iterations remaining in the loop.

Figure 15 depicts the best-case loop analysis algorithm, which is for the most part analogous to the worst-case algorithm described in the previous subsection. As a preliminary step, the algorithm computes the number of required and non-required iterations for each path, as was done in worst case. The rest of the algorithm consists of two phases. The first phase finds the shortest path *P* for the first iteration. For the first iteration only, the timing analyzer treats all first misses as misses and all first hits as hits when analyzing the cache behavior of all the paths' instructions. The major issue for selecting the shortest path *P* is determining which paths are eligible to be selected. If the loop has at least one non-required iteration, then *P* may be chosen from any of the continue paths. However, if

```
/* 1.  calculate required and non-required 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.
pipeline_info = NULL.
/* 2.  process the first iteration of the loop  */
first_miss_treatment = miss.
first_hit_treatment = hit.
IF req_iters < N THEN
   Find shortest path P among the paths in which
      P->req_iters + P->nonreq_iters > 0  &&  P->set.maxiters > 0.
ELSE
   Find shortest path P among the paths in which
      P->req_iters > 0  &&  P->set.maxiters > 0.
Concatenate pipeline_info with the current
   best-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.
/* 3.  process the remaining iterations  */
WHILE iters_handled < N DO
   first_miss_treatment = hit.
   first_hit_treatment = miss.
   IF req_iters < N THEN
      Find shortest path P among the paths in which
         P->req_iters + P->nonreq_iters > 0  &&  P->set.maxiters > 0.
   ELSE
      Find shortest path P among the paths in which
         P->req_iters > 0  &&  P->set.maxiters > 0.
   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->req_iters = 0.
   P->set.max_iters -= iters_to_do.
   P->nonreq_iters -= nonreq_iters_to_do.
   Concatenate pipeline_info with the current
      best-case union of executable paths.
   iters_handled += iters_to_do.
```

Figure 15: BCET Loop Analysis Algorithm

the loop has no non-required iterations, then *P* may only be selected from those continue paths that have required iterations.

The `WHILE-DO` loop in step 3 of Figure 15 represents the second phase of the best-case algorithm, which processes all the remaining iterations of the loop after the first. Note that the timing analyzer treats a function as a loop with a single iteration, so its best case analysis will only perform the first phase of this algorithm. In the second phase, all first misses are treated as hits and all first hits are treated as misses. In other words, the instruction cache behavior is assumed not to change during the last $n - 1$ iterations. The reason for the difference in how the worst-case and best-case loop analysis algorithms are organized is due to the classification of first hits in BCET, which can only hit on the first iteration of a loop [17]. The method of selecting the shortest path P is the same as in the first phase. Once P is selected, it is necessary to calculate the number of iterations to account for path P, which is done in the same manner as in the worst-case loop analysis. The timing analyzer will use P for all of its required iterations, plus the minimum of P's non-required iterations, P's set's maximum iterations remaining and the remaining non-required iterations of the loop. Since the method of selecting the shortest path for the best-case algorithm is analogous to selecting the longest path in the worst-case algorithm, the correctness argument for best case would also follow analogously from the worst-case explanation given above.

5. Results

The authors selected programs where the execution paths were constrained by dependencies on data values to evaluate the effectiveness of detecting and exploiting branch constraints. The programs used to assess the timing analyzer effectiveness are depicted in Table 10. The *Sumoddeven*, *Sumnegpos*, and *Summidall* programs correspond to the examples illustrated in Figures 2, 3, and 4, respectively. The *Des* program contains a loop in which the index variable is being compared to constants, giving rise to iteration-based constraints. 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

Table 10: Test Programs That Are Constrained by Dependencies on Data Values

Name	Description or Emphasis
Des	Encrypts and decrypts 64 bits
Expint	Computes an exponential integral.
Frenel	Computes noncomplex Fresnel integrals.
Gaujac	Computes the abscissas and weights of a 10 point Gauss-Jacobi quadrature formula.
LU	Performs LU Decomposition on a 100x100 matrix
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.

different paths depending upon the specified iteration of a loop. The *LU* program contains some nested loops in which the the body of the inner loop may or may not be entered based on a condition in the outer 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 six programs in Table 10 can be found in *Numerical Recipes in C* [20, 21].

The results of evaluating these programs are shown in Table 11. For each program a direct-mapped instruction cache configuration containing 8 lines of 16 bytes was used.[4] It was assumed that cache hits required one cycle, cache misses 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 [16, 11, 5]. The *Observed Cycles* represent the cycles required for an execution with worst-case input data.[5] The number of cycles was measured by enhancing a traditional cache simulator [22] to perform pipeline simulation [23]. The *Estimated Cycles* under the headings *Without Branch Constraint Analysis* and *With Branch Constraint Analysis* indicate the number of cycles estimated by the timing analyzer without and with using branch constraints, respectively. The *Estimated Ratio* is the *Estimated Cycles* divided by the *Observed Cycles*. Note that an estimated ratio of 1.0 represents a perfect prediction. Thus, the estimated ratio of 1.014 in the worst-case analysis is over 156 times more accurate than 3.192 (2.192 / 0.014), which was obtained when the branch constraint analysis was not used. Likewise, the estimated ratio of 0.969 in the best-case analysis is over 8 times more accurate.

The results show that exploiting branch constraint information in a timing analyzer can significantly tighten the WCET and BCET 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 programs. The timing analyzer was able to determine that the longer path of each program could only be executed for one half of the iterations, which reduced the overestimations. In the case of *Sumoddeven* in best case, the compiler originally determined that the loop had a minimum number of iterations of 1, but the timing analyzer

[4] A large cache would not be interesting because the test programs would fit into cache and all of the misses would be compulsory misses. A small cache shows that the timing analyzer can predict less-trivial caching behavior. Likewise, having a line size greater than the size of a single instruction tests the ability of the timing analyzer to detect hits due to exploiting spatial locality.

[5] 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 11.

Table 11: WCET and BCET Prediction Results of the Test Programs

Name	WCET Timing Prediction Results				
	Observed	Without Branch Constraint Analysis		With Branch Constraint Analysis	
	Cycles	Estimated Cycles	Estim. Ratio	Estim. Cycles	Estim. Ratio
Des	149,706	172,509	1.152	167,165	1.117
Expint	58,217	1,293,290	22.215	58,289	1.001
Frenel	47,749	48,887	1.029	47,783	1.001
Gaujac	786,786	790,116	1.004	787,134	1.000
LU	23,055,832	23,572,337	1.022	23,444,562	1.017
Sprsin	28,339	28,664	1.011	28,404	1.002
Summidall	15,340	18,090	1.179	15,341	1.000
Summinmax	16,080	17,080	1.062	16,080	1.000
Sumnegpos	11,067	13,068	1.181	11,068	1.000
Sumoddeven	15,093	16,112	1.068	15,102	1.001
Average	2,418,421	2,597,715	3.192	2,459,093	1.014

Name	BCET Timing Prediction Results				
	Observed	Without Branch Constraint Analysis		With Branch Constraint Analysis	
	Cycles	Estimated Cycles	Estim. Ratio	Estim. Cycles	Estim. Ratio
Des	65,615	22,247	0.339	57,920	0.883
Expint	125	118	0.944	118	0.944
Frenel	181	172	0.950	172	0.950
Gaujac	45,270	44,566	0.984	45,127	0.997
LU	12,883,939	637,365	0.049	11,847,472	0.920
Sprsin	17,436	17,379	0.997	17,379	0.997
Summidall	15,340	8,072	0.526	15,312	0.998
Summinmax	13,080	13,062	0.999	13,062	0.999
Sumnegpos	9,067	9,049	0.998	9,049	0.998
Sumoddeven	94	63	0.670	94	1.000
Average	1,305,015	75,209	0.746	1,200,571	0.969

was able to predict that the loop was required to iterate twice, using the methods described in the previous section. The result of this analysis was an exact BCET prediction. *LU* also showed a dramatic tightening in its BCET prediction. There were three nested loops in which the timing analyzer was able to exploit iteration-based constraints. The previous version of the timing analyzer assumed that the inner loop in these three nests would always be avoided along the best-case path of their respective surrounding loops. But in fact these loops execute on all but one iteration of the surrounding loops. 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 *Des*, *Gaujac* and *Summidall* programs that indicated that certain paths could only be executed in specific iterations. There was little WCET 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. In best case, the timing analyzer was able to determine that the loop's shortest path in *Summidall* could execute at most once, and its second shortest path could execute for at most 250 of the 1,000 iterations. Even the longest path was required to execute for at least 499 iterations. These iteration-based constraints significantly tightened *Summidall*'s BCET prediction. Similarly, *Des* contained an inner nested loop with five paths, and an iteration-based constraint required the longest path to be executed on 12 of its 16 iterations. 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 branch constraint significantly reduces the WCET overestimation of *Expint*.

Several factors contributed to the remaining WCET overestimations and BCET underestimations. First, the *Des* program in particular had several arrays in which the elements are hard-coded in the data segment, and these array element values affect various comparisons. These branch constraints were not detected in the compiler. Second, as mentioned in previous work [17], in worst case analysis some instructions conservatively categorized as misses actually hit in cache due to the order in which paths were executed because of dependencies on data values. Similarly, in best case analysis some instructions were conservatively classified as hits even though they actually miss in cache. Third, there were some minor limitations to the timing analysis that result in conservative predictions. For instance, the program *LU* had non-rectangular loop nests where the number of iterations is rounded to an integer, since the timing analyzer is designed to deal with an integral number of iterations [5, 6]. Also, the underestimation in *LU* was partially due to the fact that an iteration-based constraint was not generated by the compiler for a condition containing a complex expression that needed to be expanded. Finally, there were slightly conservative predictions that resulted from instruction caching categorizations that change between loop levels and their interaction with the pipeline analysis, affecting both WCET and BCET [11].

Table 12 shows execution time in seconds required to make WCET and BCET predictions for the test programs for the previous and current versions of the timing analyzer.[6] The times were obtained by calculating for each program the average of the elapsed times of ten executions of the timing analyzer on an UltraSPARC. The overall decrease in elapsed time for the analysis was the result of two factors. 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.

Table 12: Analysis Overhead Results of the Test Programs

Name	Seconds Required for Analysis		
	Previous Analysis Time	Current Analysis Time	Time Ratio
Des	2.155	1.422	0.660
Expint	0.374	0.293	0.783
Frenel	0.455	0.298	0.655
Gaujac	3.220	3.692	1.147
LU	1.357	1.253	0.923
Sprsin	0.132	0.129	0.977
Summidall	0.070	0.070	1.000
Summinmax	0.076	0.063	0.829
Sumnegpos	0.058	0.048	0.828
Sumoddeven	0.055	0.060	1.091
Average	0.795	0.733	0.889

6. Future Work

There are several additional aspects of using branch constraints in timing analysis that can be investigated. Many branch constraints were not detected due to function calls separating effects and the branches affected. These

[6] The response times given in Table 12 are greater than those given in our previous work [24]. In the previous work, we were only calculating a program's WCET, while in this paper the timing analyzer is computing both the WCET and BCET of each program.

branch constraints could be detected using inter-procedural analysis. Similarly, inter-procedural analysis could also detect more loop iteration constraints, in the case where one loop contains a call to a function and another loop is in the called function [25]. Further branch constraints could also be obtained from analyzing values assigned to global variables and arrays. In addition, the branch constraint analysis described in this paper could be used by a tool to ignore infeasible paths during software testing.

7. Conclusions

This paper has described how branch constraints were automatically detected by a compiler and exploited by a timing analyzer. We described techniques to efficiently detect effects that can cause the outcome of a branch to become known and detect ranges of iterations associated with branch outcomes. We presented algorithms that show how branch constraints were used to constrain the minimum and maximum iterations associated with each path in a loop and how these path constraints were used in WCET and BCET loop analysis. Finally, we showed results from a number of test programs whose worst-case and best-case paths were constrained by dependencies on data values. These results indicate that detection and exploitation of branch constraints can significantly tighten both the WCET and BCET timing predictions. While branch constraints cannot be as fully exploited using a path constraint approach as compared to a more general ILP or symbolic interpretation approach, the authors found that almost all of the constraints from a variety of application programs could be effectively used. Furthermore, the approaches used for detection and exploitation of branch constraints were shown to be quite efficient and are fully automated, requiring no interaction from the user.

8. 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] C. A. Healy, R. van Engelen, and D. B. Whalley, “A General Approach for Tight Timing Predictions of Non-Rectangular Loops,” *WIP Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 11-14 (June 1999).

[7] 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).

[8] J. Patterson, “Accurate Static Branch Prediction by Value Range Propagation,” *Proceedings of the SIGPLAN ’95 Conference on Programming Language Design and Implementation*, pp. 67-78 (June 1995).

[9] C. A. Healy, M. Sjödin, V. Rustagi, and D. B. Whalley, “Supporting Timing Analysis by Automatic Bounding of Loop Iterations,” *Real-Time Systems*, pp. 121-148 (May 2000).

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

[11] 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).

[12] Y. Hur, Y. H. Bae, S. S. Lim, S. K. Kim, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim, “Worst Case Timing Analysis of RISC Processors 1995: R3000/R3010 Case Study,” *Proceedings of the Sixteenth IEEE Real-Time Systems Symposium*, pp. 308-321 (December 1995).

[13] L. Ko, N. Al-Yaqoubi, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. Harmon, “Timing Constraint Specification and Analysis,” *Software Practice & Experience*, pp. 77-98 (January 1999).

[14] 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).

[15] F. Mueller, *Static Cache Simulation and Its Applications*, PhD Dissertation, Florida State University, Tallahassee, FL (August 1994).

[16] 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).

[17] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon, “Bounding Pipeline and Instruction Cache Performance,” *IEEE Transactions on Computers* **48**(1) pp. 53-70 (January 1999).

[18] 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).

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

[20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing,* Cambridge University Press, New York, NY (1988).

[21] 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 (1992).

[22] 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).

[23] C. A. Healy, *Predicting Pipeline and Instruction Cache Performance,* Masters Thesis, Florida State University, Tallahassee, FL (1995).

[24] C. A. Healy and D. B. Whalley, “Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints,” *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pp. 79-88 (June 1999).

[25] R. Bodik, R. Gupta, and M. Soffa, “Interprocedural Conditional Branch Elimination,” *Proceedings of the SIGPLAN ’97 Conference on Programming Language Design and Implementation*, pp. 146-158 (June 1997).