A General Approach for Tight Timing Predictions of Non-Rectangular Loops

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

Abstract

Static timing analyzers need to know the number of iterations associated with each loop in a real-time program so accurate timing predictions can be obtained. The number of iterations of non-rectangular loops vary due to dependencies on counter variables of outer loops. These loops have long presented a problem for timing analyzers since the resulting timing predictions are typically quite loose. This paper presents a general and efficient method for obtaining tight timing predictions of such loops. The total number of iterations executed by an inner loop inside a loop nest can be expressed in terms of summations. Equations representing such loops can be efficiently solved given that certain restrictions are met. We outline an approach for formulating the summations representing the total number of iterations of a loop, a method for solving the equation containing the summations, and a technique for integrating this method into an existing timing analyzer.

1. Introduction

Calculating accurate timing predictions requires knowing the number of iterations that will be performed by the loops in a program. Under certain conditions some timing analyzers can automatically determine the exact number of loop iterations [1]. Unfortunately, the number of iterations of a non-rectangular loop varies since it depends on the values of counter variables from outer loops. The worst-case execution time (WCET) and best-case execution time (BCET) predictions for a non-rectangular loop are typically quite loose. In fact, these predictions may indicate that a program does not meet its timing constraints, when it actually does.

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

The remainder of this paper has the following organization. First, we introduce related work on calculating the number of iterations executed by a loop. Second, we describe the general method for formulating the number of loop iterations as summations. Third, we present our approach for integrating this method into an existing timing analyzer. Finally, we give the conclusions for the paper and discuss an extension of the implementation.

2. Related Work

Recent work has used abstract interpretation [2] and symbolic execution [3, 4] to automatically derive the number of loop iterations. These approaches are quite powerful, but effectively requires simulating all paths of a loop for every loop iteration. Thus, they require significant analysis overhead, which would be undesirable when analyzing long running programs.

Static techniques have also been developed to bound the number of iterations of loops. One method was developed for dealing with a non-rectangular loop by calculating the average number of iterations executed each time the loop is entered [1]. This method only dealt with the special case when the number of loop iterations was dependent on a single counter variable of an immediately enclosing loop. A much more general approach to deal with non-rectangular loops is presented in this paper.

Our research was inspired by the work of Sakellariou [5, 6]. He calculated the total number of iterations for loops which are dependent on counter variables of outer loops in order to obtain better load balance by assigning approximately the same number of loop iterations to each processor. The approach used was to formulate summations representing the number of loop iterations by hand and to interface to a mathematical package offline to solve the equations. In this paper, we describe an approach to automatically calculate the average number of times that a loop will iterate during the timing analysis of a program and to use this information to obtain tighter timing predictions.

3. Formulating the Number of Iterations

The number of iterations of a single loop, where the loop variable is incremented by one (unit stride), can be represented by summations when the upper bound (b) is greater than or equal to the lower bound (a) as shown in Equation 1.

$$I = \sum_{i=a}^{b} 1 = \begin{cases} b - a + 1 & \text{if } a \le b \\ 0 & \textit{otherwise} \end{cases} \tag{1}$$

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

In general, every loop iteration count cast as a summation should evaluate to zero if the lower bound is greater than the upper bound. However, it is not always possible to evaluate the test when the bounds are symbolic. For example, consider the loop nest in Figure 1. The inner loop is a zero-trip loop for values of `i` greater than 2. We define a *partially zero-trip* loop to be a loop that is zero-trip depending on values of index variables of outer loop(s). By applying Equation 1, the iteration count of the partially zero-trip loop can be defined as shown in Figure 1. Clearly, the result is $I = 3$. However, a naive evaluation without the test results in $I = -7$. It is known that the detection of zero-trip loops in the general case is NP-complete, because it amounts to solving an LP problem. In principle, partially zero-trip loops can be normalized by changing the lower and upper bounds. This adjustment of the bounds is similarly NP-complete. The normalization process can be performed with the Fourier-Motzkin (FM) elimination method [7]. However, one can argue that real-world algorithms rarely exhibit (partially) zero-trip loops, because algorithms with partially zero-trip loops are deemed to be inefficient.

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

$$I = \sum_{i=1}^{7} \begin{cases} 3 - i & \text{if } i \le 2 \\ 0 & \textit{otherwise} \end{cases}$$

Figure 1: A Partially Zero-Trip Loop

Figure 2 shows how two different loop nests can be formulated in terms of summations. The total number of iterations to be executed by the innermost loop in each loop nest are calculated by solving the corresponding equation.

The Bernoulli formula shown in Equation 2, where $p \ge 1$ & $n \ge 1$ and B_k is a Bernoulli number, can be used to evaluate terms in a summation.

$$\sum_{i=1}^{n} i^p = \frac{1}{p+1} \sum_{k=0}^{p} \binom{p+1}{k} B_k (n+1)^{p-k+1} \tag{2}$$

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

$$\begin{aligned} I &= \sum_{i=1}^{98} \sum_{j=i+1}^{99} 1 \\ &= \sum_{i=1}^{98} (\sum_{j=1}^{99} 1 - \sum_{j=1}^{i} 1) \\ &= \sum_{i=1}^{98} (99 - i) \\ &= \sum_{i=1}^{98} 99 - \sum_{i=1}^{98} i \\ &= 4,851 \end{aligned}$$

(a) Loop Nest from Sort Program

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

$$\begin{aligned} I &= \sum_{j=1}^{100} \sum_{i=j}^{100} \sum_{k=1}^{j-1} 1 \\ &= \sum_{j=1}^{100} \sum_{i=j}^{100} (j-1) \\ &= \sum_{j=1}^{100} (\sum_{i=1}^{100} (j-1) - \sum_{i=1}^{j-1} (j-1)) \\ &= \sum_{j=1}^{100} (\sum_{i=1}^{100} j - \sum_{i=1}^{100} 1 - \sum_{i=1}^{j-1} j + \sum_{i=1}^{j-1} 1) \\ &= \sum_{j=1}^{100} (102j - j^2 - 101) \\ &= 102 \sum_{j=1}^{100} j - \sum_{j=1}^{100} j^2 - \sum_{j=1}^{100} 101 \\ &= 166,650 \end{aligned}$$

(b) Loop Nest from LU Decomposition Program

Figure 2: Deriving the Total Number of Iterations for Two Loop Nests

We can represent summations with nonunit strides, where the stride s is specified along with the lower bound a and upper bound b. Equation 3 shows how a nonunit stride can be used in a conventional summation, where e is an expression and $e[i \leftarrow si + a]$ denotes the substitution of all free occurrences of i by $si + a$. This way, summations with strides can be represented by uniform summations.

$$I = \sum_{i=a}^{b,s} e = \sum_{i=0}^{\lfloor (b-a)/s \rfloor} e\ [i \leftarrow si + a] \tag{3}$$

Summations with nonunit strides are more difficult to evaluate since one has to deal with summations of floors. Equation 4 shows how a floor can be converted to an expression involving a modulo operation. A modulo operation can often be simplified using Equation 5. Equations 1-5 can be used to correctly determine that the total iterations for the loop nest in Figure 3 is 1,717.

$$\left\lfloor \frac{n}{m} \right\rfloor = \frac{n - n\%m}{m}, \text{if } m > 0 \;\&\; n > 0 \tag{4}$$

$$\sum_{i=0}^{n} (i\%d)^p = \begin{cases} \sum_{i=0}^{n} i^p, \text{if } n < d \\ \sum_{j=0}^{\lfloor n/d \rfloor} \sum_{i=0}^{d-1} i^p + \sum_{i=0}^{n\%d} i^p, \text{if } n \geq d \end{cases} \tag{5}$$

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

Figure 3: A Loop Nest Containing a Nonunit Stride

4. Implementation

The implementation for evaluating the summations described in the previous section was accomplished by using the General-Purpose Algebraic Simplifier (GPAS) portion of the Ctadel system [8, 9]. The authors' timing analyzer [10] and Ctadel were compiled separately, but Ctadel is directly integrated into the timing analyzer by linking the object files. This avoids unnecessary overhead that would result from passing expressions between the timing analyzer and GPAS through a software bus. The summations are formulated in the timing analyzer and GPAS is invoked as a C function with the summation parameters as arguments.

The timing analyzer attempts to verify that there are no zero-trip loops for an inner loop by expanding its initial value and limit. Likewise, the timing analyzer determines if there are no partially zero-trip loops in the loop nest. However, if the verification is inconclusive, the loop nest may or may not contain (partial) zero-trip loops. For instance, consider the loop nest in Figure 4. The expansion of the innermost loop initial value (`i`–3) yields the range [–3..6]. Expanding the limit expression (`j`+8) gives [8..18]. The timing analyzer is able to guarantee that the inner loop is not zero-trip since the initial value is never greater than the limit.

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

Figure 4: Innermost Loop Detected Zero-Trip Free by the Timing Analyzer

Now consider the loop in Figure 5. Expanding the initial value gives [0..8], while the limit is [0..9]. Since these ranges overlap, the test is inconclusive. However, the loop nest is not zero-trip due to the $j<i$ condition in the middle loop. Since the range analysis can be used to safely verify if a loop is partially zero-trip, it is possible to use the results in deciding which summation solver to use. For example, the loop in Figure 4 can be safely cast into a summation without a bounds tests, while the summations for the loop in Figure 5 requires a bounds test.

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

Figure 5: Innermost Loop Nest Detected Zero-Trip Free by GPAS

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

(1) GPAS evaluates the summation while testing the bounds of the index variables.

(2) GPAS evaluates the summation without testing for bounds.

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

The algorithm for selecting the appropriate method is described in Figure 6. The exact solutions are computed using safe assumptions in the possible presence of partially zero-trip loops, using either method (1) or (2). This algorithm will resort to method (3) only in the presence of multiple loops with nonunit strides.

```
The timing analyzer verifies that the loop nest is not
      (partially) zero-trip.

IF the check is successful THEN
      The loop nest is formulated into summation without
            bounds tests and presented to GPAS.
ELSE
      The check is inconclusive and the loop nest is cast
            into a summation with bounds tests.
      The rewritten summation is presented to GPAS.

IF GPAS is able to solve the summation THEN
      RETURN the integer count.
ELSE
      GPAS could not solve the summation in the presence
            of two or more loops with nonunit strides.
      RETURN conservative bounds on the sum.
```

Figure 6: Algorithm for Selecting a Summation Method

The following approach is used in the timing analyzer to obtain tight predictions of non-rectangular loops. The timing analyzer calculates WCET and BCET predictions based on the maximum and minimum number of iterations for a non-rectangular loop, respectively. These predictions are made in case a user requests the WCET or BCET predictions for the loop. In addition to these absolute predictions, the timing analyzer also calculates *average* WCET and BCET predictions for each loop. During the course of analyzing a program, the timing analyzer invokes GPAS to find the number of iterations of each loop in the nest. To calculate the average number of iterations for a loop, we divide the total iterations by the total number of times the loop is entered. The timing analyzer remains quite efficient overall, since in nearly all realistic cases GPAS can sum loop iterations in well under one second.[1]

As an example, consider the innermost loop from the *sort* program in Figure 2 which has 4,851 total iterations. We also calculate the number of times the current loop is entered by calculating the total number of iterations for the loop that encloses the current loop. In this example, the loop is entered 98 times. Thus, the average number of iterations for the loop is 49.5 (4,851/98). The average number of iterations is used to calculate the average WCET and BCET predictions. When a non-integer is calculated, we round up for the WCET prediction and truncate for the BCET prediction since our loop analysis algorithm requires an integral number of iterations.

5. Conclusions and Ongoing Work

In this paper we have outlined a general approach to accurately bound the number of iterations of a non-rectangular loop nest. The timing analyzer formulates a summation expression, evaluates this sum, and then computes the average number of iterations of the innermost loop to tightly bound the WCET and BCET.

One extension currently under study is handling the general case of a nested loop having arbitrary nonunit strides. Our current implementation only addresses the more common cases of nonunit strides: for example, loop nests containing an unlimited number of nonunit strides, provided that all loop lower and upper bounds are constant. In [6] a "splintering" technique is suggested to compute exact summation results. However, the process of splintering results in a large number of subterms containing similar summations. The number of sums is proportional to the product of the stride values. We are working on a more efficient implementation in which the number of sums is proportional to the LCM of the stride values.

[1] The authors have created a Web page demonstrating the functionality of the GPAS. It can provide the number of loop iterations for a predefined example loop nest or one entered by the user. The URL is `http://www.cs.fsu.edu/˜engelen/iternum.cgi`.

6. Acknowledgements

The platform for calculating the symbolic number of iterations of a loop was established in part with assistance of Viresh Rustagi and Mikael Sjödin. The conversations with Rizos Sakellariou about his work on calculating the total number of iterations for a loop nest was invaluable.

7. References

[1] 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).

[2] 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).

[3] 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).

[4] Y. Liu and G. Gomez, “Automatic Accurate Time-Bound Analysis for High-Level Languages,” *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 31-40 (June 1998).

[5] R. Sakellariou, *Symbolic Evaluation of Sums for Parallelising Compilers*, Wissenschaft & Technik Verlag, Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics (1997).

[6] R. Sakellariou, *On the Quest for Perfect Load Balance in Loop-Based Parallel Computations*, PhD Dissertation, Department of Computer Science, University of Manchester, Manchester, England (October 1996).

[7] M. J. Wolfe, *High Performance Compilers for Parallel Computers*, Addison-Wesley, Redwood City, CA (1996).

[8] R. van Engelen, L. Wolters, and G. Cats, “Ctadel: A Generator of Multi-Platform High Performance Codes for PDE-based Scientific Applications,” *Proceedings of the 10th ACM International Conference on Supercomputing*, pp. 86-93 (May 1996).

[9] R. van Engelen, L. Wolters, and G. Cats, “Tomorrow’s Weather Forecast: Automatic Code Generation for Atmospheric Modeling,” *IEEE Journal of Computational Science and Engineering* **4**(3) pp. 22-31 (September 1997).

[10] 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).