Simplifying WCET Analysis By Code Transformations

Hemendra Singh Negi Abhik Roychoudhury Tulika Mitra
School of Computing, National University of Singapore
{hemendra,abhik,tulika}@comp.nus.edu.sg

ABSTRACT

Determining worst case execution time of a program by static analysis is important for the design of real-time software. WCET analysis at the programming language level requires the detection of the longest path in the program. A tighter bound on the WCET of a program can be achieved by identifying the infeasible paths in the program's control flow, which is a difficult problem. Due to the branches in a program structure, the number of possible paths in the program can grow exponentially. In this paper we present a method to transform the code such that the number of paths in the program could be reduced and hence the search space for the infeasible paths is brought down. This could reduce the complexity of determining infeasible paths in a program and also result in tighter WCET.

1. INTRODUCTION

The design of real-time embedded software requires that a guarantee must be given about the time taken by a program. Especially, in hard real-time systems, where the failure of a program to give results within a required amount of time may have serious consequences, the problem of determining Worst Case Execution Time (WCET) of a program becomes more critical. The WCET of a task is also important for scheduling the tasks in real-time systems. However, it is very difficult to obtain an accurate WCET of a task. Therefore, a tight bound on the WCET by static analysis methods is always desired to achieve better scheduling of tasks.

The problem of determining the WCET of a program by static analysis methods has to be solved at the following two levels [12]: (1) Programming language level, to discover the longest path from the start to the end of the program [7] and (2) Micro-architectural level, to take into account the effect of features such as pipeline, cache and branch prediction [6, 5]. The determination of WCET at the programming language level involves the detection of infeasible paths in the program and then use that information to give a tight bound on the execution time of the task ([3, 11]). In this paper, we only consider the programming language level analysis of the WCET. We will first describe the types of infeasible paths along with some techniques on how to detect them. We then present our idea to reduce their numbers and get a better estimation of the WCET of a task.

The knowledge about infeasible paths in a program can be used to give a tighter bound on the WCET. There could be infeasible paths because of the correlation between branches. For example, in Figure 1(A), <3,4,5,6> is an infeasible path because if the outcome of branch at line number 3 is true

```
1 for(i:= 0; i<limit; i++)
2 {
3      if ( i < 3 )
4            S1;
5      if ( i > 3 )
6            S2;
7 }
        (A)
```

```
1 sumeven := 0;
2 for (j:=0; j<=limit; j++)
3 {
4    if (j % 2 == 0) then
5       sumeven = sumeven + j;
6 }
        (B)
```

Figure 1: Infeasible paths due to branch correlation

then the outcome of branch at line number 5 can not be true. Detection of such types of infeasible paths has been studied in [2, 3]. Another type of infeasible paths which can be present in a program are ones that span over multiple iterations of a loop. For example consider the code to calculate the sum of even numbers, as shown in Figure 1(B). If the path <3,4,5,6> is taken in some iteration of the loop then it is not possible to take it again in the next iteration of the loop.

Detection of infeasible paths in a program is an important but difficult problem. A technique to detect and use infeasible path information is presented in [3]. We briefly describe their technique here to motivate how it could be benefited by our code transformation approach. In [3], the authors have used an effect based technique to determine the infeasible paths in a program and used this information for calculating the WCET of a loop. They first determine how a conditional branch can be effected by an assignment to a variable and/or the outcome of another conditional branch. The conditional branch could have one of the three types of effects: *unknown, fall-through* or *jump*. The effects on the conditional branches by the assignment of a variable are then exploited while traversing the basic blocks in every path of the program to determine whether the path is feasible or not.

Timing prediction of loops via control flow (as in [3]) poses a lot of problems for timing analyzer. A lot of space is required to represent all the paths, unavailability of which might abort the timing analyzer. Moreover, a large number of paths will result in a significant increase of the execution time of the timing analyzer. Therefore, a method which can reduce the number of paths, will be very useful. We present our approach as a pre-processing step to reduce the number of paths and hence reduce the complexity and time taken by the timing analyzer.

2. OUR PROPOSED TECHNIQUE

```
1 x = 0; t = 1;
2 for (i = 0; i < 10; i++)
3 {
4     if ( x == 0)
5         S1;
6     else
7         S2;
8     if ( x == 2)
9         t = -1;
10    if ( x == -1)
11        t = 1;
12    x = x + t;
13 }
```

Original Code

(A)

```
1 x = 0; t = 1;
2 for (i = 0; i < 10; i++)
3 {
4     if (x == 0)
5         S1;
6     else
7     {
8         S2;
9         if (x == 2)
10            t = -1;
11        else
12            if( x == -1)
13                t = 1;
14    }
15    x = x + t;
16 }
```

Code after loop path reduction

(B)

Figure 2: Example code to illustrate our technique

We observe that the detection of infeasible paths is inherently exponential in terms of the number of branch constraints. Hence, we try to develop a strategy to identify which branch conditions can be removed from consideration during the detection of infeasible paths such that the complexity of the detection algorithm could be reduced and at the same time a tighter bound on the WCET could be provided. We also try to optimize the code such that the number of paths in the code can be reduced. We try to exploit the constraints generated at branch conditions to optimize the code. In this section we will illustrate our technique with the help of an example and also show how the WCET analysis as per [3] can be benefited by it.

Reducing number of loop paths. Consider the piece of code shown in Figure 2(A). The values of x in the Figure 2(A) seen at line number 4 are in the form of a simple harmonic motion around the value 0. The sequence of values seen for x at line number 4 are (0,1,2,1,0,-1)*. '*' represents zero or more repetitions. The control flow graph for the code in Figure 2(A) is shown in Figure 3(A). From Figure 3(A), it is apparent that there are 3 branch conditions and 8 paths in each iteration of the loop. The various possible paths for each iteration in terms of basic blocks executed are given below.

```
a : 2 3 4 6 7 8 9 10 11      b : 2 3 4 6 7 8 10 11
c : 2 3 4 6 8 9 10 11        d : 2 3 4 6 8 10 11
e : 2 3 5 6 7 8 9 10 11      f : 2 3 5 6 7 8 10 11
g : 2 3 5 6 8 9 10 11        h : 2 3 5 6 8 10 11
```

However, it could be observed from the branch constraints that the results of branch conditions at block 3 and 6 could never be true simultaneously. Therefore block 4 can never be executed together with block 7. Moreover, both (true/false) paths from block 3 reaches block 6 and 8 where block 6 is a conditional statement and blocks between 6 and 8 could only be executed along with the false path from block 3. Also the constraint variable (x) of block 6 does not get assigned along the true path from block 3. Therefore, blocks 6 and 7 could be moved in the false path from block 3. Figure 3(B) shows the result of such a transformation.

Due to the transformation, the number of paths in the loop gets reduced to 6 from the initial number 8. Using the similar observation for conditional branches at blocks 3 and 8, the code can be optimized as shown in Figure 3(C), reducing the number of paths to 5. And finally the code can be modified to as shown in Figure 3(D), reducing the number of paths to 4.

The WCET analysis on the basis of the technique given in [3] will involve the following steps: determining the effect of assignments on the three branch conditions and then using this information to determine the infeasible sequence of paths. The technique will be greatly benefited by the optimization as the number of paths are decreased and so is the complexity of the technique which traverse over the paths to determine feasibility of paths and also the sequence of paths which is infeasible in consecutive iterations.

```
1  x := 0; t := 1;
2  for (i := 0; i < 9; i++)
3  {
4     if ( x == 0)
5        S1;
6     else
7     {
8        S2;
9        update (x , t);
10    }
11    x = x + t;
12 }
```

```
update (x , t)
{
  switch (x)
  {
     case 2 : t = -1; break;
     case -1: t = 1; break;
     default: t = t;
  }
}
```

Figure 4: Example code after path length equalization

Equalizing path lengths. The optimization given in the previous section will transform the original example code into an optimized code as shown in Figure 2(B). We now try to deduce a transformation for this code to further simplify the WCET analysis. For our purpose, we propose a new type of block in the CFG along with basic blocks. The new block will be called as *functional block* which will represent a function. The various paths inside such a functional block will not be considered in the WCET analysis. We will see later in this section that a safe WCET bound can still be reached even though the number of paths considered for WCET are reduced without actually removing such paths.

We can identify the following paths, in each iteration of loop, from Figure 3(D).

```
a : 2 3 4 10 11              b : 2 3 5 6 8 10 11
c : 2 3 5 6 8 9 10 11        d : 2 3 5 6 7 10 11
```

The execution of loop will result in the following sequence of taken paths (abdbac)*. It is apparent that *aa, bb, cc, dd* along with *ad, abc, bdc* and many more, are infeasible sequences of paths that could be taken in consecutive iterations. Determining such infeasible sequences of paths with techniques as in [3] will be quite complex and computationally expensive. However, we propose the following code transformation to simplify things. The code in Figure 2(B) can be modified to the code as in Figure 4. The CFG for the modified code is shown in Figure 5

The combining of the blocks in path from 6 to 10 into *update* function and writing the *update* function in the way shown in Figure 4 could be very fruitful. Every call of the *update* function will take a constant amount of time due to the structure of the *update* function, hence the time taken

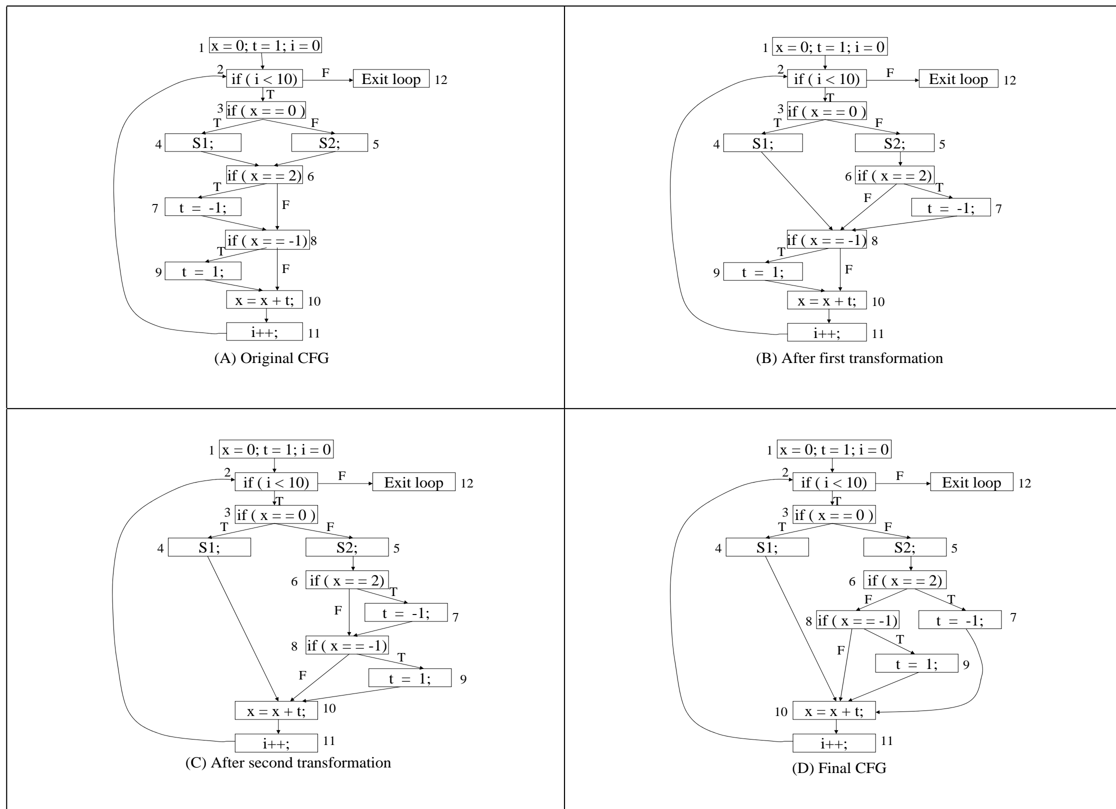

Figure 3: Reduction of number of loop paths in Control Flow Graph

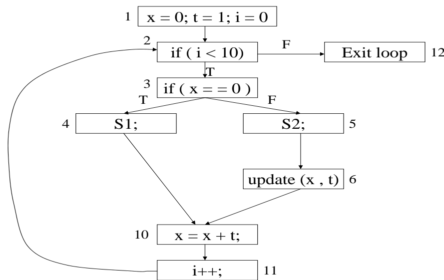

Figure 5: Control Flow Graph after path equalization

to execute block 6 in Figure 5 will always be the same, irrespective of the path taken within the function. The block 6 is a *functional* block in Figure 5, and a constant time can be assigned to it just like basic blocks.

The transformation of code will result in the following two possible paths (Figure 5) in each iteration of loop, that should be considered by the analyzer to detect infeasible sequences of paths taken in consecutive iterations.

```
a : 2 3 4 10 11          b : 2 3 5 6 10 11
```

The execution of loop will result in the following sequence of taken paths (abbbab)*, from which it is easy to identify that the infeasible sequences of paths are *aa, bbbb, abba, ababa.* The transformation results in reducing the search space for possible infeasible paths, to a great extent. Therefore the complexity of infeasible path detection as per the technique in [3] is greatly reduced and will result in a tight and safe bound on WCET. Even though there exists other infeasible paths when the paths inside the update functions are considered, such infeasible paths can be ignored in WCET analysis as every call to update function takes constant amount of time.

3. CONCLUSION

Detection of infeasible paths in a program is important for WCET analysis. However, it is difficult to detect all the infeasible paths in a program and moreover the search space for infeasible paths could grow exponentially in terms of number of branches in the program. Our technique can not only reduce the number of paths in the program by optimization but could also consolidate a group of paths into one path as far as WCET analysis is concerned. Thus we reduce the complexity of infeasible path detection while still maintaining the safe bound on WCET.

4. DISCUSSION & FUTURE WORK

Mueller and Whalley in [8] have also exploited the idea of restructuring the control flow and replicating code. However, they have used it for compiler optimization via avoiding conditional branches. Previously, Puschner in [9, 10]

```
#include <stdio.h>
main() {
    int i, j;
    printf ("enter a number: ");
    scanf ("%d", &i);
    if (i == 1)
        i = i+1;
    if (i == 2)
        i = j;
    if (i == 3)
        i = i+3;
    if (i == 4)
        ++i;
    if (i == 5)
        printf (" i = %d\n",i);
    if (i == 6)
        printf (" j = %d\n",j);
}
```

(A)

```
#include <stdio.h>
main() {
    int i, j;
    printf ("enter a number: ");
    scanf ("%d", &i);
    f1(i);
    f2(i);
}
```

```
f1(i){
  switch (i){
    case 1: i= i+1;
            break;
    case 2: i = j+0;
            break;
    case 3: i = i+3;
            break;
    case 4: i = i+1;
            break;
  }
}
```

```
f2(i){
    switch (i){
      case 5:
        printf (" i = %d
                \n",i);
        break;
      case 6:
        printf (" j = %d
                \n",j);
        break;
    }
}
```

(B)

Figure 6: Example Code: Toy6

have also given a code transformation based approach to reduce the complexity of WCET analysis. The author has proposed a single path paradigm for programs so that there could only be a single path in a program hence making WCET determination simple. Such a transformation will have to trade a lot of performance with predictability. On the other hand, with our proposed technique, the WCET analysis complexity could be reduced to a large extent without much trade off in performance. Another work by Al-Yaqoubi ([4, 1]) also describes a technique to simplify the control flow of complex loops by partitioning the control flow into sections that are limited to a predefined number of paths. Each section is then treated by the timing analyzer as a loop that iterates only once. Using the same example **Toy6** as in [1] (shown in Figure 6(A)), we see that our transformation (shown in Figure 6(B)) can reduce the number of paths in Toy6 from 64 to 1, without much increase in the code length and still giving a tight prediction for time using timing analyzer as in [3]. Function `f1` in Figure 6(B) can be assigned a constant amount of time (equal to any single case of the switch statement), similarly function `f2` can also be assigned a constant amount of time and both `f1, f2` are treated as functional block while calculating WCET. Hence, our approach can reduce the complexity of control flow much better than that in [4], without trading of much in terms of code length and tightness of estimation.

Note that, our technique requires a modification in the actual code in order to reduce the complexity, which in case of some hard real-time systems might not be allowed. It should also be noted that our technique is not a timing analysis technique. It could be used as a **preprocessing** step to other infeasible path detection and timing analysis techniques such as [3, 2]. Our technique could reduce the complexity of other techniques and provide tighter bounds on WCET. Other techniques need to be modified in order to handle the functional blocks. However, at the present stage we do not have a concrete technique to determine the potential regions in the code which could be worked upon for transformation. For example, a certain type of *if* structures in the program can be optimized for reducing the paths as in the given example in this paper and also a group of basic blocks can be converted into a functional block by transforming *if* statements into a *switch* statement inside the new function. In our future work, we plan to come up with efficient methods to automatically determine potential regions for transformation.

5. REFERENCES

[1] N. Al-Yaqoubi. Reducing timing analysis complexity by partitioning control flow. Master's thesis, Florida State University, Tallahassee, FL, 1997.

[2] R. Bodik, R. Gupta, and M. Lou Soffa. Refining data flow information using infeasible paths. In *ESEC/SIGSOFT FSE*, 1997.

[3] C.A. Healy and D.B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions on Software Engineering*, 28(8), 2002.

[4] L. Ko, N. Al-Yaqoubi, C. Healy, E. Ratliff, R. Arnold, D. Whalley, and M. G. Harmon. Timing constraint specification and analysis. In *Software Practice and Experience*, pages 77–98, January 1999.

[5] X. Li, T. Mitra, and A. Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. In *Design Automation Conference (DAC)*, 2003.

[6] Y. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1995.

[7] Y-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM Design Automation Conf. (DAC)*, 1995.

[8] F. Mueller and D. B. Whalley. Avoiding conditional branches via code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 55–66, June 1995.

[9] Peter Puschner. Transforming execution-time boundable code into temporally predictable code. In *Proceedings of IFIP World Computer Congress, Stream on Distributed and Parallel Embedded Systems*, pages 163–172, 2002.

[10] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, January 2002.

[11] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, pages 132–140, 2001.

[12] Reinhard Wilhelm. Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone. In *VMCAI 2004*, volume 2937 of *LNCS*, pages 309–322, 2004.