CODE INSPECTION

SHYAAMKUMAAR KRISHNAMOORTHY
EURECA EXCHANGE STUDENT
880106-P691
DEPARTMENT OF COMPUTER SCIENCE
MALARDALEN UNIVERSITY

SUPERVISED BY CHRISTER SANDBERG

7th July, 2009

ABSTRACT

Real time systems, used in most of the day-to-day applications, require time critical execution of tasks. Worst Case Execution Time Analysis (WCET) is performed to ensure the upper bound on the time they can take to execute. This work aims to perform a static analysis of the industry standard code segments to provide valuable information to aid in choosing the right approach towards WCET analysis. Any code segment can be analyzed syntactically, to gain some insights to the effects that a particular coding syntax format may have. With focus on the functions and looping statements, valuable information regarding the code segments inspected can be obtained. For this purpose, the code segments from CC-Systems, Västerås, were inspected to obtain more information related to this. Scope graphs generated by SWEET, the Swedish Execution Time Tool, were extensively used to aid this work. It was found that syntactical analysis could be performed effectively for the code segments that were analyzed as a part of this task.

ACKNOWLEDGEMENTS

I owe a great deal to my teachers, friends and members of my family who have encouraged, supported and enlightened me throughout this work.

First and foremost I thank all the members behind the EURECA project (www.mrtc.mdh.se/eureca<http://www.mrtc.mdh.se/eureca> funded by the Erasmus Mundus External Cooperation Window (EMECW) of the European Commission project) who provided me with an experience of my lifetime. I would like to express my heartfelt gratitude to Dr. Sasikumar Punnekkat for making this entire program hassle-free and for the support he has provided me during the past six months.

I would like to acknowledge the debt I owe to my supervisor Christer Sandberg without whose guidance and support this thesis work would not have been what it is today. He has been guiding me patiently since the day I decided to take up this thesis work. I also thank all the other members of WCET research team for their support.

I also thank CC Systems, Västerås for providing me the code segments for this work. I specially thank Mr. Anders Oberg for his support.

My four years of study at Amrita School of Engineering, Coimbatore has provided me with the opportunity to learn and spend time with a lot of wonderful people. Many thanks goes to all my teachers at the Department of Computer Science who have taught me over the years. I thank my HOD, Professor P.N Kumar for his motivation.

My deepest gratitude goes to my family for believing in me.

TABLE OF CONTENTS

1 . INTRODUCTION

1.1 CODE INSPECTION

Code inspection can be regarded as the static analysis of existing program segment to determine the factors that may help in choosing the best possible approach towards measuring the Worst Case Execution Time. The word static here means that the code is analyzed during compile time and not during run time on a particular underlying hardware. This means that the program segment is inspected for some factors which may possible affect the time taken for a program to execute. Some of the key factors may be

- Variables that affect the number of loop iterations and their computations
- The data dependencies influencing the loop exit conditions
- The extent of usage of functions

This list will serve as the set of preliminary findings motivating this thesis work. Moreover, this thesis work will hold its focus on industrial code segments for analysis.

With a detailed summary of the effect of these factors, the best possible approach may be suggested to the tools that compute the Worst Case Execution Time.

1.2 EARLIER WORKS

A research paper on the same subject has been presented at the WCET workshop at the Euromicro Conference by Christer Sandberg [Christer]. This research paper forms the basic framework for my thesis work.

1.3 PLATFORM AND HARDWARE SPECIFICATIONS

This thesis work uses the following hardware configuration and all the findings are with respect to this particular system configuration.

Operating System - Linux 2.6.27.19-170.2.35.fc10.i686 i686
System - Fedora release 10 (Cambridge)
Processor - Intel(R) Core(TM)2 CPU T5600
Total Memory - 2.0 GB

2 . WORST CASE EXECUTION TIME ANALYSIS (WCET)

Worst Case Execution Time Analysis can be defined as the analysis of a program code to determine the upper bound of the time it takes to execute [And03]. This upper bound or the worst case time can depend upon several factors. First and foremost, the underlying hardware plays a significant role in this calculation and in this research thesis, this will be kept a constant. The underlying hardware assumptions are explicitly stated in the forthcoming pages. With the hardware constraint fixed, the next pivotal factor affecting the worst case time will be the properties of the code such as the nesting extent of loops and functions, presence of complex loops etc. For instance, the presence of functions and loops and their nested forms are the key areas of interest as they affect the flow of the program and in such cases a flow analysis would reveal interesting statistics. However, this thesis work focuses on preliminary analysis of certain measurements on the code which may suggest the best possible WCET analysis approach on that code.

2.1 DYNAMIC WCET ANALYSIS

Dynamic Analysis can be described as the analysis done based on the measurement of the running time of a particular program segment on a particular hardware. Since this thesis work mainly deals with static analysis, further details on dynamic analysis will be beyond the scope of this thesis work.

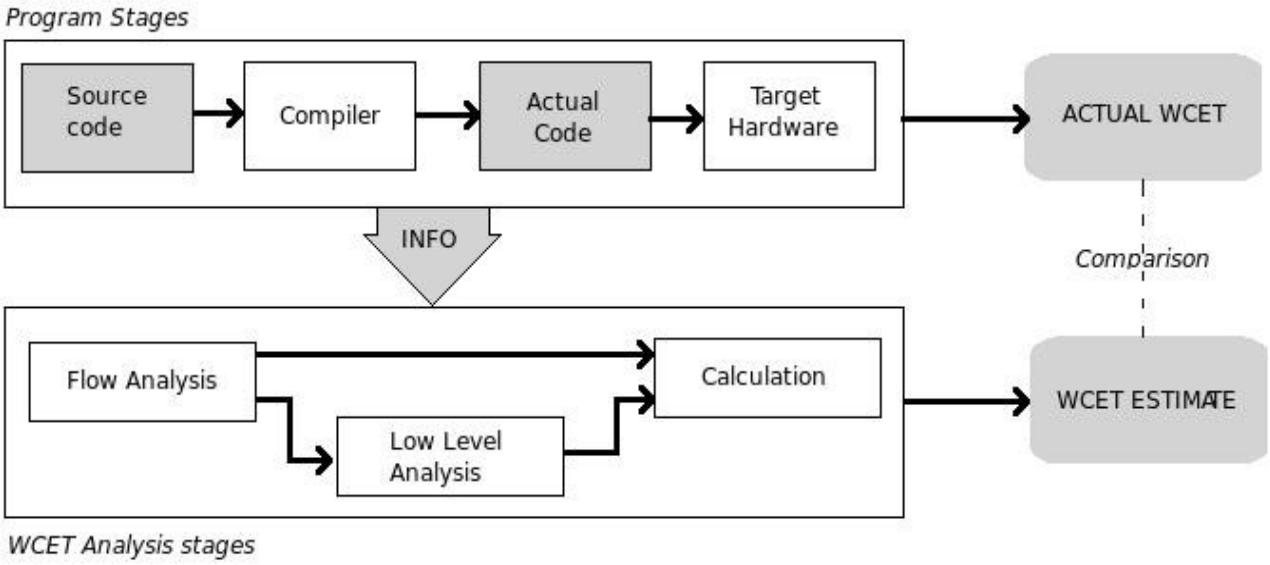
2.2 STATIC WCET ANALYSIS

Static WCET analysis deals with the detailed inspection of the factors that affect the Execution time. However, the word 'static' emphasizes the face that analysis is made during compile time rather than run time. Thus a variety of factors that may affect the flow of execution of program are studied and their effects analyzed. Flow analysis is one of the pivotal steps in static WCET analysis.

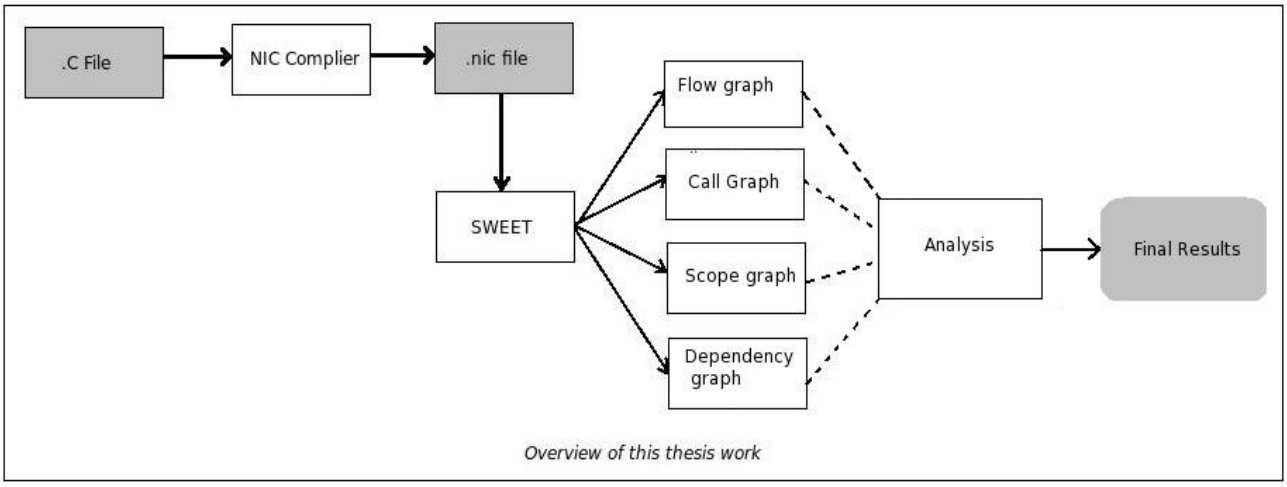
Flow Analysis can be done in several ways like Abstract Interpretation , Syntactical Analysis etc
To be precise, Abstract Execution, based on Abstract Interpretation can be used to determine the loop bounds through extensive analysis which makes it quite heavy. On the other hand, Syntactical Analysis can determine the same by matching predefined syntactical patterns of the program model used by the analysis tool such as SWEET.

3 . TOOLS

Andreas Ermedahl[And03] provides the components of the WCET analysis process as shown.



In this thesis work, the source code is examined to provide suggestions to the WCET tool and so the focus will be towards the source code part. The following diagram may provide an overview of this thesis work and provide a better understanding of the working of the tools used.



3.1 NIC COMPILER

NIC stands for New Intermediate code. This particular compilation tool actually consists of a front-end named lcc[1] and a back-end that is plugged into lcc to produce an intermediate format of the file with an extension .lni.

1 lcc is a retargetable compiler for Standard C. It generates code for the ALPHA, SPARC, MIPS R3000, and Intel x86 and its successors. [prince]

3.2 SWEET

SWEET is an acronym for Swedish Execution Time tool and has been developed by Mälardalen WCET research group[2]. SWEET performs flow analysis on the input code and aids in WCET estimation. Although SWEET can perform low-level analysis, for the purpose of this thesis work, it will be mainly used to produce the following set of graphs : *Flow graphs, Call graphs ,Scope graphs* and *Dependency graphs*. These graphs are then used to determine the measurements of this thesis work.

3.3 FLOW GRAPH

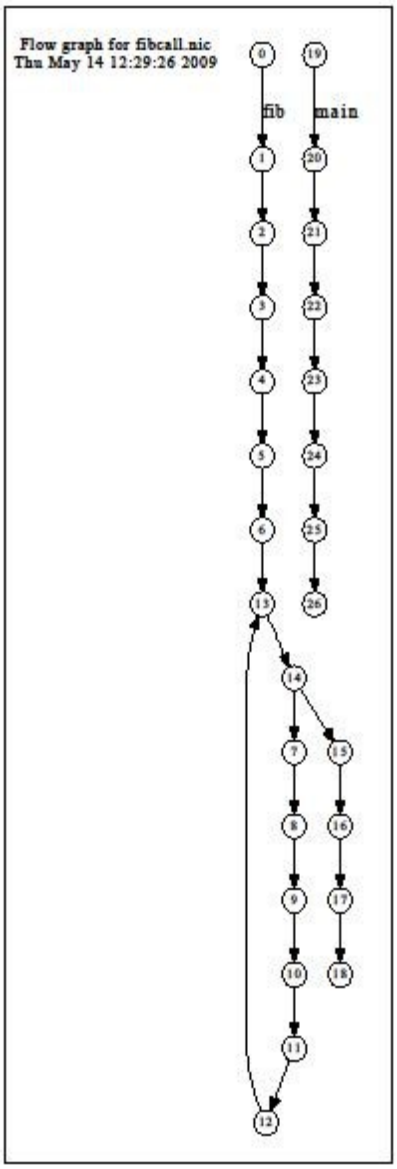
A flow graph(or rather a control flow graph) is a directed graph that depicts the flow of execution of program statements. Consider the example shown in Figure 3.3

```
int fib(int n)
{
 int  i, Fnew, Fold, temp,ans;

   Fnew = 1;  Fold = 0;
   for ( i = 2; i <= n; i++ )
   {
    temp = Fnew;
    Fnew = Fnew + Fold;
    Fold = temp;
   }
   ans = Fnew;
 return ans;
}

int main()
{
 return (fib(5) + fib(30));
}
```

Figure 3.3

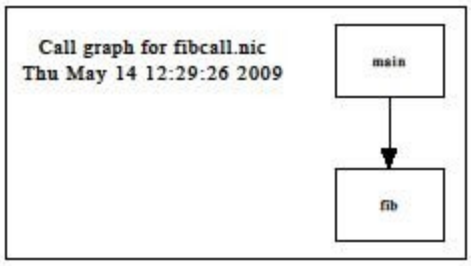
The flow graph for this example will be as shown.

2 http://www.mrtc.mdh.se/projects/wcet/sweet.html



3.4 CALL GRAPH

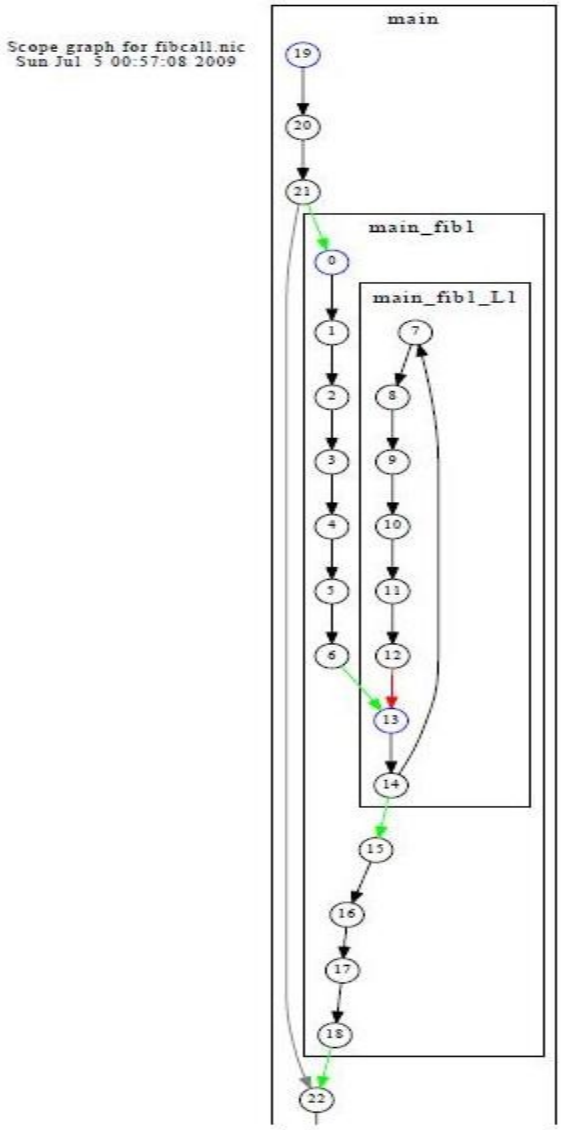
The sequence of execution of functions is shown in a call graph. There is an edge from the calling function to the function being called. In case of recursive function, there is an edge to itself. For the example in figure 3.3, the call graph will look like

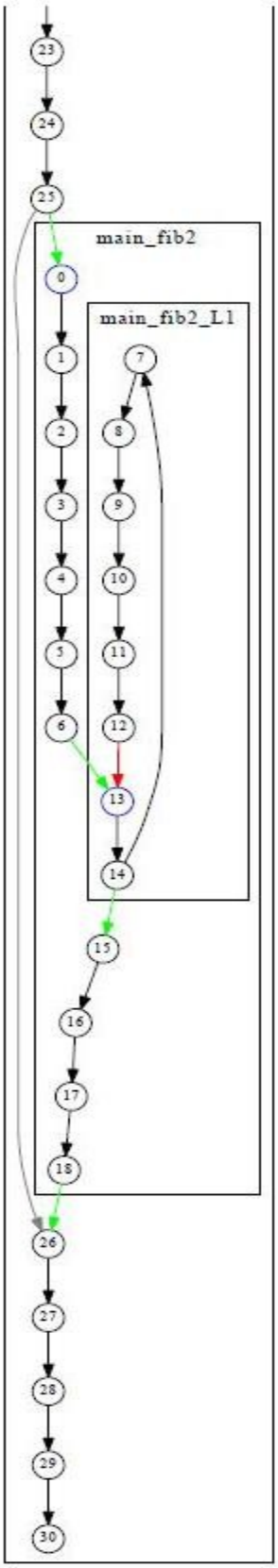


3.5 SCOPE GRAPH

Scope graph is a directed graph generated by SWEET. A scope graph can clearly illustrate the flow representation as it uses the basic blocks to represent the nodes. The term 'scopes' are used frequently with respect to a scope graph and these may represent repeating or differentiating loops, functions (recursive functions included) and code segments that are non-reducible. Formal definition and more information on the same can be found in [And03]. The main aim of using a scope graph is to elucidate the context sensitive nature.

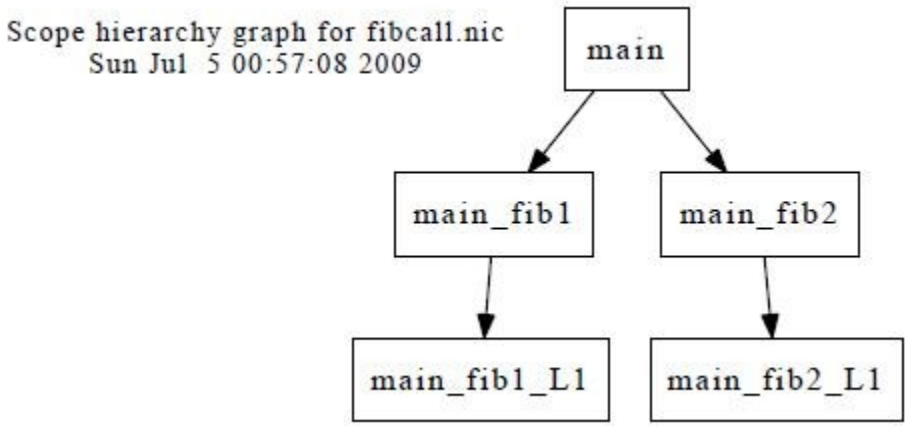
To illustrate the scope graph clearly, consider the example in figure 3.3. The scope graph generated by SWEET will look as follows.




23
24
25
main_fib2
0
main_fib2_L1
1
7
2
8
3
9
4
10
5
11
6
12
13
14
15
16
17
18
26
27
28
29
30

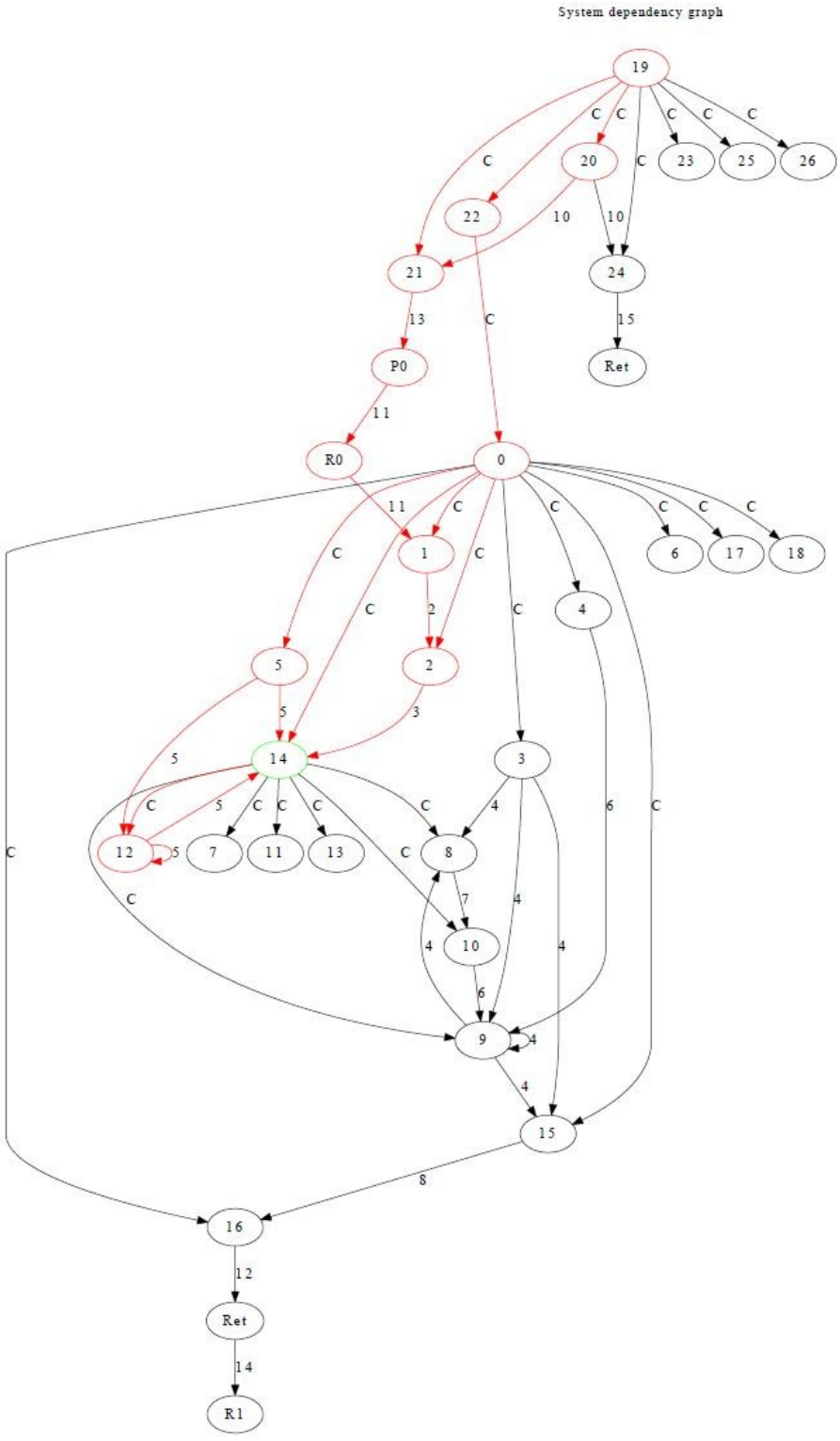
X

From the graph generated, it can be seen that each loop, function and their nesting is illustrated in the graph. For instance, it can be seen that the function main() has a scope wider than fib() and the two calls from main to fib are also illustrated with the unique scope labels as main-fib1 and main_fib2. A scope hierarchy graph is also generated which gives out the hierarchy (a tree structure)among the scopes in a program. It must be noted that 'scopes' are nodes in this graph. The scope hierarchy graph for the above mentioned example is as follows.



3.6 DEPENDENCY GRAPH

SWEET also generates a system dependency graph which is a union of both control and data dependency graphs. A dependency graph represents the dependencies between entities in a program. In this particular case, data dependency graphs were of particular interest and they were used to analyze the dependencies from loop exits. An outline of how a dependency graph(representing both control and data dependencies) will look like for the example mentioned in figure 3.3 is as follows.


System dependency graph
19
20
22
21
24
23
25
26
P0
Ret
R0
0
1
6
17
18
4
5
2
14
3
12
7
11
13
8
10
9
15
16
Ret
R1

4. CODE INSPECTION

Inspection of code segments were carried out with respect to the following measurements. It was done with a aim of finding interesting information on loops. This work concentrated on obtaining knowledge about the following : Variables that are used in loops, Global Variables, Data dependencies and Extent of usage of functions.

4.1 ANALYSIS OF THE DATA DEPENDENCIES FROM LOOP EXITS TO GLOBAL VARIABLES

A situation in which an instruction is dependent on a result from a sequentially previous instruction before it can complete its execution[DAINTITH]. Data dependency can be of three types : True dependency, anti-dependency and output dependency. True dependency is said to exist if an instruction is dependent on some other instruction that precedes it. When a statement uses a value which is altered after its execution, anti dependency is said to exist. Presence of an anti dependency can result in inability to alter the order of execution of statements. If the order of execution can affect the output of the statements, then output dependency is said to exist. Figure 4.1 clearly illustrates this.

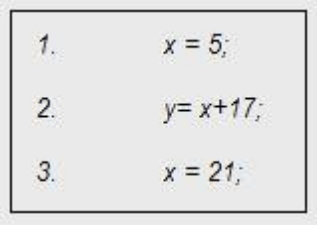


Figure 4.1

In this example, statements 2 is truly dependent on statement 1 as it uses 'x', defined at statement 1. Statements 2 and 3 are anti-dependent because the value 'x' is altered in 3 and so their execution has to be in the same order. In this case, output dependency is also exhibited as the change in order of execution will change the outcome of these statements.

Data dependencies from loop exits to global variables are of interest particularly in cases where they obtain their values from *external inputs*[3], thus making them analogous to a loop variable whose value is runtime variant. In such cases, a bound on the values it can take must be provided apriori to the tool that performs analysis to get a valid loop bound value. Since in a static analysis the program is not subjected to execution, the presence of such variables are analyzed to check for the dependencies that are present. By analyzing the data dependencies present we can obtain two important results. First and foremost, we can check for possibilities of existence of code motion. i.e we can check if there are any invariant loops present in the code[AUS86]. This can be achieved by tracing the data dependencies. If a particular branch exists without any dependency, it indicates the presence of a code segment which is invariant.

3 External Inputs here can be an input from a hardware or a value from a task running in parallel.

Secondly, we can check a loop for the dependencies between loops. i.e. we can find out if a certain loop's number of iterations may vary between the iterations of an outer loop, by checking if any exit condition in the inner loop is data dependent on a statement that is non-invariant with respect to the outer loop.

4.2 LOOP ANALYSIS

The loops present in the code segments when analyzed, can provide us with interesting information about the code. A typical loop contract will be

<initial value, condition check, update on the loop variable >

A structure like this is simple to match provided the initial value is a constant and there is only one loop variable that is updated. Moreover the condition should not be too complex an expression.

For example, a loop like : *for(init = 0; init <10; init++)* is simple to analyze.

With the introduction of another variable(that is updated), the extent of complexity increases.

```
for(init = 0; init<10; init++, counter--)
{
…
if(counter >5)
break;
}
```

In this case the value of the variable *counter* too is an important consideration. Furthermore, if this variable is present in outer loop such as this

```
for(a = 15; a >= 10; a--)
{
        counter = a+15;
        for(init = 0; init<10; init++, counter--)
        {
        …
        if(counter >5)
        break;
        }
}
```

It gets worse when functions and nesting between them are involved. Hence a careful analysis of dependencies is required to gain insight about the loops.

As explained above, the dependencies from loop exits are a key measure and are obtained by

analyzing the data dependency graph. In addition, a loop is checked for its simplicity. 'Simplicity' here refers to a classification of loops based on the presence of a few properties. A loop can be regarded simple if any of the following conditions occur.

- There are no dependencies to non-invariant pointer values or global variables
- Number of variables updated by a single decrement or increment in the loop = Number of loop variables
- Number of conditionally updated loop variables = 0
- The number of variables whose data is altered in other loops = 0

This is an important measure as this can aid in suggesting the approach to be taken. For simple loops, a pattern matching approach can be followed while complex loops can be tackled with Abstract Execution[Tools].

One more important factor to be noted is the use of pointer variables. Presence of pointers brings out a whole new problem when they are used to hold addresses. An example (see figure 4.2) will throw some light upon this important issue.

```
int x,y,.*p;

p = &x;

while ( p != 0)
{
 ...

 }
```

Figure 4.2

Problem to be addressed in this type of case is how will the analysis tool interpret the comparison between *p's* value and an integer.

4.3 EXTENT OF USAGE OF FUNCTIONS

The extent of usage of a function here is resulted out as a ratio between the number of nodes representing a function in a scope graph to the number of nodes in a *call graph*[4]. Since each scope graph clearly illustrates the number of scopes of each block ; i.e. each and every block that is a looping construct or a function call is depicted pictorially, the trace of the flow can be studied extensively as it is sensitive in nature. A call graph on the other hand, provides the overall context insensitive flow of the program. Thus a comparison of the number of scope graph nodes (of the type 'function') and the number of nodes in a call graph can depict the extent of usage of the functions present in the code.

4 Source - Ryder, B.G., "Constructing the Call Graph of a Program," Software Engineering, [RYDER]

5 CODE INSPECTION AT CC SYSTEMS

For the purpose of gaining evidence for this thesis work, inspection of industrial code segments seemed pivotal. The aim was to subject as much industrial code as possible for analysis. So, a set of project files (c files) written for Automating the welding process was analyzed during the period of 3 weeks at CC Systems,Västerås . Code inspection was carried out on these project files and they were analyzed with regard to the measurements carried out for this thesis work.

As a part of thesis work, the code segments (ESAB code constructs) provided by CC Systems, Västerås , were made to run through SWEET to obtain a few interesting parameters. Main objective of this thesis work was to determine how much information can be obtained on loops that are present in an industry standard code. To aid this, the SWEET tool was appended with a couple of new functions that could analyze the loops present, in detail. The scope graphs were generated for these industrial code segments by SWEET. SWEET however, was not used to perform a full fledged static flow analysis. Instead it was used to generate the the graphs used to analyze the properties of the code.

5.1 CODE SEGMENTS

Industry standard code segments had these characteristics

- Minimal loops were found and they were mostly *simple*.
- Generous use of 'switch case' construct
- Occasional occurrences of loops which were waiting for an interrupt to occur.
- Conditional statements were generously used and a few operations were used as the basic blocks on which other functions were developed on.

In this set of files, since they were developed to achieve the functionality of automated welding, most of the file segments were heavily dependent on basic functions for controlling the '*wire feeder*[5]'.

5.2 CHALLENGES

5.2.1 Coding Standards

First and foremost problem encountered was that of coding standards. Since SWEET required the codes to be present in ANSI C standard, the code segments had to be manually edited to follow this standard.

5.2.2 Presence of specific loops

Loops were mainly used to make the system wait until an external event/interrupt occurred. Compilation of these forms of loops through NIC compiler leave the CFG disconnected. SWEET does not analyze disconnected CFGs.

Figure 5.1 illustrates one such loop. This problem was solved by introducing the variable as shown in

5 'wirefeeder' was used to control the rate at which wire was fed to the automatic welding process.

figure 5.2 thus taking advantage of the fact that SWEET does not carry out *constant propagation optimization*[6].

```
while (1)
{

HW_SET_BIT = 1;

/* wait until interrupt occurs  */

}
```

Figure 5.1

```
while (CONDITION_TRUE)
{

HW_SET_BIT = 1;

/* wait until interrupt occurs  */

}
```

Figure 5.2

5.2.3 Presence of target-specific functions

Target specific statements and function calls were also found in the code segments that were analyzed. These function calls had to be replicated in some cases and in most cases, they had to be omitted from analysis.

Occasional cases of 'SWEET crashes' were observed. This was due to unreachable code segments. In such cases, they were either altered or commented out.

5.3 RESULTS

Total number of Lines	15566
Average extent of usage of functions	1.945
Total number of loops	19
Total number of exit conditions	19(one per loop)
Non invariant variables	34
Outer loop dependencies	0
Linearly updated loop variables	38 (max = 3, min = 0 per loop)
Exit conditions depending on pointer values	0
Conditionally updated loop variables	4 (max 1, min 0 per loop)
Dependencies to global variables	0

Table 5.1

6 Constant propagation optimization is the technique in which the value of a particular variable is substituted in its place to achieve optimization.

6 OUTCOME OF THE ANALYSIS

From the table 5.1, it can been seen that analysis was very much constrained due to the minimal occurrences of simple loops and absence of pointers. Looping statements had one exit condition and utmost 3 linearly updated variables. i.e. Variables that are were updated either by a single increment or decrement within the loop body. In total 38 of these were found. No dependency from loop exit conditions to global variables were observed and only 4 instances of conditionally updated variables were present.

However, manual observation revealed the **excessive** usage of conditional statements like 'Switch Case' and 'if' constructs and the code relied heavily on target specific functions. So this provided very little scope for extensive loop analysis. Although most of the functions were target specific and exhibited signs of nesting, recursive functions were not found.

Interestingly, the presence of 'masked' break statements as shown in figure 5.3 were found. A syntactical analysis tool looking for certain patterns must be aware of this 'masked break' to succeed in computing an upper bound for such loops. Note that a conditional assignment of the loop variable 'i' seen in isolation can not guarantee that *{ i = 0, i < bound_1, i ++ }* gives a safe upper loop bound of 'bound_1'. Although in this special case of conditional assignment it is safe.

```
for ( i =0; i< bound_1; i++)
{
  ....
  ....
  if(....)
   {
    ...
    i = bound_1;        // 'MASKED' BREAK STATEMENT
   }
}
```

Figure 5.3

Although there is a little evidence about the occurrence of non-trivial loops, they seem to contradict with the manual inspection of the code. This may be largely attributed to the optimization of the code carried out by NIC compiler and the lack of proper documentation (about NIC compiler) hinders any possible conclusion that can be made about them.

7 CONCLUSION

Presence of a large number of conditional statements such as 'Switch' and 'if' suggests that there may have been patterns for infeasible path analysis. However this finding came up only upon manual observation of the code segments. By then it was too late to set up an analyzing framework for them.

The absence of pointer value dependencies and dependencies to outer loops indicate that an efficient syntactical analysis could be performed effectively which would provide an upper bound on the loops that were encountered. This makes abstract execution unnecessary thus reducing the amount of time taken to perform the flow analysis. This would make the flow analysis effective.

However it must be noted that the above mentioned conclusions were a result of the inspected code segments only and not much generalized conclusion can be provided as the inspection was carried out on a limited amount of code.

8 FUTURE WORK

The constraint on time limited the number of files(15566 LOC) that could be analyzed within the short span of three weeks. However there is certainly a scope to find more interesting information and the following could be regarded as possible tasks that could be carried out in the future.

- New framework could be set up to analyze conditional statements such as 'if' and 'switch'
- Inspection of more such industrial code segments for the same properties as they would help in comparison.

9 BIBLIOGRAPHY

[Christer] Christer Sandberg. Inspection of Industrial code for syntactical loop analysis. WCET workshop, Euromicro Conference,http://www.mrtc.mdh.se/index.php?choice=copyright&url=http://www.mrtc.mdh.se/publications/0827.ps&id=0827

[Tools] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenstr¨om, P. 2008. The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embedd. Comput. Syst. 7, 3, Article 36 (April 2008), 53 pages. DOI = 10.1145/1347375.1347389 http://doi.acm.org/10.1145/1347375.1347389

[1] Evaluation of Automatic Flow Analysis for WCET Calculation on Industrial Real-Time System Code, Dani Barkah *(Volvo CE, Eskilstuna, Sweden)*, Andreas Ermedahl, Jan Gustafsson, Björn Lisper, Christer Sandberg, 20th Euromicro Conference of Real-Time Systems, (ECRTS'08), Prague, Czech Republic, July, 2008

[And03] *A Modular Tool Architecture for Worst Case Execution Time Analysis, Ermedahl* Andreas, Acta Universitatis Upsaliensis, Uppsala 2003, Uppsala dissertations from the faculty of Science and Technology 45. 200pp. Uppsala. ISBN 91-554-5671-5

[JanAnd08] *Merging Techniques for Faster Derivation of WCET Flow Information using Abstract Execution*, Jan Gustafsson and Andreas Ermedahl, Mälardalen Real-Time Research Center (MRTC) Västerås, Sweden, http://www.artist-embedded.org/docs/Events/2008/WCET/wcet2008_gustafsson.pdf

David Schmidt, Kansas State University - Abstract interpretation and static analysis , International Winter School on Semantics and Applications Montevideo, Uruguay, 21-31 July 2003

[AUS86] Aho, Alfred V. , Sethi Ravi, Ullman Jeffey D, Compilers, principles, techniques and tools , Addison Wesley Publications,1986

[PRINCE] http://www.cs.princeton.edu/software/lcc/

[RYDER] Ryder, B.G., "Constructing the Call Graph of a Program," Software Engineering, IEEE Transactions on , vol.SE-5, no.3pp. 216- 226, May 1979

[DAINTITH] JOHN DAINTITH. "data dependency." A Dictionary of Computing. 2004.*Encyclopedia.com.* (July 1,2009). http://www.encyclopedia.com/doc/1O11-datadependency.html