Abstract

Knowing the boundaries of loops is an important prerequisite for both, static and dynamic Worst Case Execution Time (WCET) analysis. However, loop bound calculation is a complex task of its own, and often more effort than planned has to be put into it. This paper describes a simple and quick method for loop bound calculation using a model checker that cannot only find loop bounds for integer iterator variables but works with practically all kind of loops.

1 Introduction

Embedded real-time systems are quickly gaining importance. Mechanical subsystems of cars and airplanes are replaced by interconnected electronic components, which do not only replace their mechanical counterparts, but also add safety-critical and comfort functions. An interesting fact is that over the years the correct behavior in the value domain has always been an important issue while the correct timing behavior has been of lesser interest. This is not the only reason, why the breakdown causes for cars shift from the mechanical to the electronic and software domain. It is difficult to estimate how many of these breakdowns are caused by timing violations, but it is becoming evident that the execution-time behavior of applications can no longer be neglected. Today the number of available timing analysis tools is very limited but since this is currently a very active area of research, it can be expected that this will change in the near future.

To understand how the approach described in this article fits into the whole WCET analysis process, it is necessary to understand how WCET analysis tools work in general. Usually a WCET analysis tool, either static or measurement based, performs some or all of the following steps:

1. Code analysis on source or object code level, generation of control flow graph (CFG)
2. Analysis of control flow, data flow and execution paths including function call and loop analysis
3. Estimation of execution times of statements, basic blocks or paths using static analysis or run-time measurements. This may include low level analysis of cache, pipelines, branches, and other CPU features.
4. WCET calculation from measured or calculated execution times.

To allow an estimation of an upper bound for the execution time in step 4, the maximum number of iterations for each loop must be known. This information is acquired during loop-bound analysis.

---

This work has been supported by the FIT-IT research project “ATDGEN - Automatic Test Data Generation for WCET Measurements” under project number 812653/1729-GLE/BLC.
in step 2, which is an important sub-problem of timing analysis. Obtaining loop bounds can be a difficult task, especially when considering the multiple different ways loops are implemented. Therefore it is desirable to find a general applicable method for loop-bound determination.

This document describes a fast and flexible method to obtain loop bounds for general loops on source-code level for applications written in ANSI-C. The described method uses a model checker and a binary search algorithm to determine the upper bound of loops. This is implemented by using assertions on automatically generated counter variables. The implementation is part of a measurement based WCET analysis framework which is currently being developed as part of the “Automatic Test Data Generation for WCET Measurements (ATDGEN)” project.

1.1 Contributions

The main contribution of this work is to show that loop bounds can be obtained with little programming effort for applications written in ANSI-C. The performance is probably inferior to the approaches presented in Section 2 but the presented method is simple to implement and can be applied to all kind of loops.

The second contribution is to point out how complicated tasks like symbolic execution can be removed from a WCET analysis framework by using additional independent tools such as a model checker. The WCET analysis framework which uses the presented loop-bound calculation approach uses model checking also for test data generation and the detection of infeasible paths.

1.2 Structure of this Article

The next section describes state-of-the-art related work. Section 3 explains the proposed method based on an example. In Section 4 an evaluation of the proposed method using different examples is presented. The last sections give a conclusion of this work and an outlook to further research that could be performed on this topic.

2 Related Work

In this section two cutting edge loop-bound analysis publications are presented. Article [6] presents a method for deriving flow information by means of abstract execution which is a special form of symbolic execution. The proposed method can automatically calculate simple loop bounds and loop bounds for nested loops, as well as many types of infeasible paths. The “SWEET” analysis tool can detect mutually exclusive paths within loop bodies and gives very accurate results by using infeasible path information. However, there are certain limitations for the loop types. For instance the tool cannot use pointers as iteration variables, which is the way how look up tables are implemented in many automatically generated automotive applications.

Another interesting approach is presented in [4]. A data-flow analysis is performed on the object code level of the application. Based on this data-flow analysis possible loop counters and invariants are identified. Afterwards the loop tests are analyzed and combined with the results of the data flow analysis to calculate the loop bounds.

The CBMC model checker [1][3] that is used for this work is a fast and efficient bounded model checker which uses ANSI-C as its primary input language. CBMC supports almost the complete ANSI-C syntax including dynamic memory allocation, case statements, loops and function calls. When model checking is performed, all loops are unrolled until the maximum iteration count is
reached. Unfortunately there is no way to get this information directly from the model checker. By introducing an additional loop counter variable, like in the presented approach this can be achieved.

An approach that is similar to the method introduced in this article is described in [7]. The described approach uses binary search to maximize the WCET of a whole application. The proposed approach operates on object code and focuses on cache analysis, while the author states that it can be easily extended to perform other low level analysis methods like pipeline analysis as well. While loop counters are inserted to bound the maximum loop count the actual maximum of the loop counts are not known outside the model checker. In order to perform model checking a C model is generated from the basic-block graph (BBG) model of the object code. The C model contains the operations performed on object-code level, the execution times of the basic blocks as estimated by static analysis and other properties of the object code. The author states that the advantage of the approach over other static analysis tools is that the whole execution path is known by the model checker and therefore can be used for cache analysis. However this might decrease the scaleability of the proposed method since methods reducing the complexity like segmentation [9] are not performed.

3 The Model Checking Approach

This section describes the method which has been implemented within the ATDGEN WCET analysis framework to achieve loop bounds. The design goals were to develop a method that is reasonably fast and is convenient to implement. Symbolic execution requires a high implementation effort and therefore does not meet the requirements. Model checking was used within the MoDECs project [8] to generate test data and identify infeasible paths where it proved to be an efficient replacement for a custom symbolic execution engine. Since the CBMC model checker provides good performance and an ANSI-C interface a method has been developed to calculate loop bounds using CBMC.

The example which is used to explain the method is shown in Listing 1. To preserve space the modifications needed for the proposed method have already made by inserting lines 3, 10 and 14. This is currently done manually but shall be performed automatically by the WCET calculation framework in the near future. The sample application is very simple but the method has been tested on applications of up to 4200 lines and 2600 basic blocks.

The basic idea of the proposed approach is to place an additional loop counter (line 3) within each loop, which is, in the example in Listing 1, named “loopcounter”. The loop counter is incremented with each iteration of the loop (line 10). After the loop or at the end of the application an assertion is placed (line 14) which compares the loop counter to a predefined value (IMAX). Likewise to a C-compiler preprocessor definitions can be issued to CBMC using the -D command line option. When CBMC is called with the command line “cbmc -D IMAX=2 sum_500.c” the end of the model checker output shows that the assertion has failed (“VERIFICATION FAILED”). If

```
int k = 500;
int loopcounter = 0;
int main (int argc, char** argv)
{
    int i;
    int n = 1;
    for (i=0; i<=k; ++i) {
        loopcounter++;
        if (i%2) n += i;
        else n -= i;
    }
    assert (loopcounter < IMAX);
}
```

Listing 1. The sample application
IMAX is selected higher than or equal to the maximum iteration counter, the assertion at the end holds ("VERIFICATION SUCCESSFUL").

When examining the model checker output directly, messages about loop unwinding as shown in Listing 2 can be seen. Therefore it should be possible to acquire the loop bound directly. However, there are two reasons why this should not be done: First, if there is more than one loop it will be unknown, how the unwinding messages relate to the loops in the source code. Is loop \( n \) the \( n \)th loop in the source code or the \( n \)th loop reached during execution? Second, this feature is not documented in the CBMC manual [1] and the proposed approaches should not rely on it.

The loop counter can only be compared to a single value at a time. As a result a binary search algorithm has to be used: IMAX is doubled until the assertion holds. Afterwards IMAX is decreased in the same manner until \(|IMAX - IMAX_{n-1}| = 1\). The minimum value for which the assertion holds is tracked at the same time and eventually holds the maximum loop counter at the end of the process.

The search algorithm is currently implemented using a simple Perl script. Listing 3 shows a typical search run. The first column displays the examined application. Second, the current maximum loop count is shown. The next column shows if a counter example has been found ("CE +", IMAX too low) or if the checked model is safe ("CE +", IMAX \( \geq \) max. iterations). The last column displays the time required for model checking. When the script is finished, the bottom line shows the maximum number of iterations, in case of the sample this is 501.

While this method is definitely slower than the analysis presented in [6] it has a big practical advance: The proposed method can be used for virtually any kind of loop. The only requirement is that it has to be possible to insert a command to increase the loop counter somewhere within loop.

We experienced that automatically generated code makes heavy use of loops iterating over arrays using pointers as iterator variables. It is difficult and requires a high implementation effort to implement this using symbolic execution, however the effort using the model checker based method is very low. The analysis framework only generates some additional code to create a loop counter and to specify an assertion to be checked. The complicated work is done by the model checker which saves much implementation effort that can be used on other parts of the WCET analysis framework.

Listing 2. Finding the Loop Bound

```
$ cbmc -D ICOUNTERMAX=502 sum_500.c
file sum_500.c: Parsing
Converting
Checking sum_500
Generating GOTO Program
Pointer Analysis
Adding Pointer Checks
Starting Bounded Model Checking
Unwinding loop 1 iteration 1
Unwinding loop 1 iteration 2
...
Unwinding loop 1 iteration 500
Unwinding loop 1 iteration 501
size of program expression:
2519 assignments
Generated 1 claims, 0 remaining
VERIFICATION SUCCESSFUL
```

Listing 3. Finding the Loop Bound

```
$ ./run.pl sum_500.c IMAX
maximising IMAX for sum_500.c
sum_500.c 1 ... CE + 0.04
sum_500.c 2 ... CE + 0.05
sum_500.c 4 ... CE + 0.05
sum_500.c 8 ... CE + 0.06
sum_500.c 16 ... CE + 0.07
sum_500.c 32 ... CE + 0.10
sum_500.c 64 ... CE + 0.17
sum_500.c 128 ... CE + 0.31
sum_500.c 256 ... CE + 0.58
sum_500.c 512 ... CE - 0.91
sum_500.c 1024 ... CE + 0.88
sum_500.c 2048 ... CE + 1.04
sum_500.c 4096 ... CE + 1.22
sum_500.c 8192 ... CE + 1.30
sum_500.c 16384 ... CE - 0.91
sum_500.c 32768 ... CE + 1.18
sum_500.c 65536 ... CE - 0.91
sum_500.c 131072 ... CE + 1.18
MAXIMUM ITERATION COUNT: 501
```

4
4 Experimental Results

In this section we evaluate the new method using more complex examples. Some of the examples shown in Table 1 are the same as in [6], which are taken from [5], allowing a direct comparison of both methods.

All analyses were done on a PC with a 2 GHz Intel Core2 Duo (only 1 core used) with 2 GB memory, running Linux 2.6.23.12. The Sweet benchmarks were run on Intel Pentium 4 @ 4GHz and 1 GB RAM on linux 2.6.9.11.[6]

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>#L</th>
<th>#BB</th>
<th>#Iter</th>
<th>T [s]</th>
<th>TMAX [s]</th>
<th>Mem [MB]</th>
<th>TSWEET [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum_500.c</td>
<td>∑(+1 -2 +3 ... +499 -500)</td>
<td>16</td>
<td>5</td>
<td>501</td>
<td>10.7</td>
<td>1.2</td>
<td>10.1</td>
<td>-</td>
</tr>
<tr>
<td>array.c</td>
<td>linear search in an pointer-array of integer (30 elements)</td>
<td>35</td>
<td>9</td>
<td>29</td>
<td>7.3</td>
<td>1.1</td>
<td>12.7</td>
<td>n.a.</td>
</tr>
<tr>
<td>test_239.c</td>
<td>calculate 239! in a loop (ignore overflow)</td>
<td>21</td>
<td>6</td>
<td>240</td>
<td>12.0</td>
<td>4.5</td>
<td>18.4</td>
<td>-</td>
</tr>
<tr>
<td>test_13247.c</td>
<td>calculate 13247! in a loop (ignore overflow)</td>
<td>21</td>
<td>6</td>
<td>13248</td>
<td>3845.6</td>
<td>277.9</td>
<td>178.7</td>
<td>-</td>
</tr>
<tr>
<td>insertsort.c</td>
<td>insertion sort (10 elem., nested loops)</td>
<td>22</td>
<td>7</td>
<td>45</td>
<td>34.0</td>
<td>4.5</td>
<td>18.3</td>
<td>-</td>
</tr>
<tr>
<td>lut.c</td>
<td>search an integer value in an array using a pointer as iteration variable</td>
<td>73</td>
<td>12</td>
<td>8</td>
<td>8.5</td>
<td>1.1</td>
<td>16.2</td>
<td>n.a.</td>
</tr>
<tr>
<td>b.s.c</td>
<td>Binary search in an array of 15 integer elements</td>
<td>114</td>
<td>34</td>
<td>4</td>
<td>0.4</td>
<td>0.1</td>
<td>5.4</td>
<td>0.01</td>
</tr>
<tr>
<td>fibcall.c</td>
<td>Iterative Fibonacci, used to calculate fib(30)</td>
<td>72</td>
<td>26</td>
<td>29</td>
<td>0.7</td>
<td>0.1</td>
<td>5.5</td>
<td>0.02</td>
</tr>
<tr>
<td>nsichneu.c</td>
<td>Simulates an extended Petri net. Automatically generated code with more than 250 if-statements</td>
<td>4253</td>
<td>2686</td>
<td>2</td>
<td>67.6</td>
<td>17.3</td>
<td>201.5</td>
<td>32.37</td>
</tr>
<tr>
<td>cover.c (L1)</td>
<td>Program for testing many paths (1th loop)</td>
<td>640</td>
<td>1298</td>
<td>120</td>
<td>25.1</td>
<td>1.8</td>
<td>8.2</td>
<td>-</td>
</tr>
<tr>
<td>cover.c (L2)</td>
<td>Program for testing many paths (2nd loop)</td>
<td>640</td>
<td>1298</td>
<td>50</td>
<td>21.2</td>
<td>1.8</td>
<td>8.1</td>
<td>-</td>
</tr>
<tr>
<td>cover.c (L3)</td>
<td>Program for testing many paths (3rd loop)</td>
<td>640</td>
<td>1298</td>
<td>10</td>
<td>14.0</td>
<td>1.7</td>
<td>7.9</td>
<td>-</td>
</tr>
<tr>
<td>cover.c (all)</td>
<td>Program for testing many paths (all loops)</td>
<td>640</td>
<td>1298</td>
<td>180</td>
<td>18.8</td>
<td>1.9</td>
<td>8.5</td>
<td>3.65</td>
</tr>
</tbody>
</table>

Table 1. Benchmark Results

Table 1 shows the name (source file) of the test [Benchmark] including a short description of the test [Description], followed by the number of source lines [#L] and the number of basic blocks [#BB]. [#Iter] gives the number of loop iterations, which were verified by manual analysis. The last example contains three loops which have been evaluated individually (cover.c L1,L2,L3) and all together (cover.c all). The next column shows the analysis time in seconds [T] for the binary search over all iteration steps (multiple runs of CBMC) while the next column [T.MAX] shows only the analysis time for the first iteration (single run of CBMC) where the assertion holds, which is the case when IMAX=#Iter+1. The next column [Mem] shows the memory usage in MB. Finally the last column [TSWEET] shows the analysis time required by the SWEET tool [6]. Test cases that are
missing but could have been performed with SWEET are marked with a dash, test cases that are not appropriate for the SWEET tool are marked with “n.a.”.

As we expected, the SWEET analysis tool performs generally faster. Additionally, it is interesting to note, that, without the iterative loop bound search, the CBMC model checker performs slightly better than the SWEET tool. When looking at the internals of the model checker, it can be noticed that loop unrolling is performed as required. Thus the model checker has internal knowledge of the loop bounds. When it can be managed to access this information directly, the analysis time would drop dramatically from $T$ to $T_{MAX}$ since the iterative search is only a means to acquire the loop bound from the model checker.

4.1 Resource Usage

Figure 1 shows the resource usage in relation to the number of iterations. To retrieve the data, the built in loop unrolling of CBMC was restricted to a certain number of iterations. The execution time (depicted with square markers) rises exponentially with the number of unrolling operations until the maximum number of iterations is reached. The memory usage (diamond markers) rises linearly with the number of iterations. When the unrolling is performed according to the maximum number of iterations the execution time and the memory usage drop and remain approximately constant. The limits depend also on the complexity of the loop body.

During the MoDECs project [8] we examined medium to large industrial embedded applications using the CBMC model checker and did not encounter any problems like exorbitant memory usage or execution times related to the code size of the applications. Therefore it can be assumed that the proposed approach will work for applications of the same size as well, even if they contain loops.

4.2 Limitations

The presented approach is limited to bounded loops. For loops iterating over dynamic structures like linked lists the data contained within the lists must be known, however it can be allocated by the application using malloc and filled with data by the application. The supported C language features are described in [2], however virtually all language features except unions are supported. As pointer arithmetic is supported, it is easily possible to find the maximum loop bound when iterating over a (zero terminated) string or a array, which is the way look up tables are implemented in automatically generated code.

5 Conclusion

In this article, we presented how model checking can be used as a practical tool to add loop bound analysis to a WCET analysis framework. While the presented approach is slower than currently available projects it has the advantage of low implementation effort and it can be used on a wide range of loop types.
The advantage of the proposed solution is that the model checker is an individual and replaceable tool with a lean interface between model checker and WCET analysis framework. The disadvantage is, that it might be desireable to have access to additional flow information generated during loop analysis like infeasible paths or mutual exclusive basic block information, which cannot be shared between model checker and WCET analysis framework.

6 Outlook

As shown in Section 4 the time needed for a single model checker run is as fast as the abstract interpretation method discussed in [6]. The next step is to find solutions to eliminate the need for searching the loop bound but to acquire it directly from the model checker. An additional challenge to this approach are programs with more than one loop since there needs to be a way to uniquely identify loops in order to assign a loop bound to a specific loop.

References