Licentiate thesis proposal

Improvements of the Flow Analysis in WCET Tools

Christer Sandberg
Department of Computer Science and Electronics
Mälardalen University, Västerås, Sweden
christer.sandberg@mdh.se

22 Feb 2005

Abstract

The worst case execution times, the *WCET*, are often essential to know for tasks that have to fulfill deadlines. Such tasks can often be found in real time systems. In order to calculate the WCET, flow constraints, like maximum iterations of loops, needs to be known. Entering all such flow constraints to a WCET tool require a lot of work for the user to be done. Therefore methods for achieving flow constraints automatically are important. However, most of today's WCET tools require manual annotations of loops for real sized programs. One of the reasons is that the most powerful analysis methods are too costly in terms of computation power.

This paper suggest three approaches to reduce the required computations to be done, in order to be able to analyze larger programs.

- Reduce the program size by removing all parts of the program that do not affect the control flow.
- Find syntactical methods with low calculation cost in order to find upper loop bounds.
- Make a detailed study of industrial code in order to find commonly used language constructs. The result of this work can be used to guide the syntactical methods, so it will be performed prior to that.

This work will result in three papers and a licentiate thesis.

1 Introduction

An approximation of WCET can be obtained either by measurements or by analysis. Measuring means to execute the program for some input and measure the elapsed time. Such an approximation will give a time that is equal to the actual WCET in case the worst case is generated by the set of input data, else lower. When analyzing the program safe approximations can be done, i.e. decisions that always leads to a longer execution time. With this approach in the analysis, the time will be at least equal to the actual WCET.

The industrial standard of today is to measure the WCET, or to estimate from manual code analysis. Measuring requires that a hardware platform or time accurate simulator is available. However, often the input data to the program can vary in such ranges that testing them all to find the one that causes WCET is not feasible. Therefore the programmer has to read the code as well as the system specifications carefully in order to find out for which combination of input data the WCET will occur. This is a tedious and error prone work. Even when it has been done, the important question "Is this the worst case?" can not be answered for sure. The same problem occurs when estimating WCET by manual analysis of the code.

To be sure to find an approximation that is guaranteed to be safe (equal to or larger than the actual WCET) we need to statically analyze the program. Current methods either require the bounds of a number of loops to be entered by the user, or too much computation to be able to work with large programs. We aim to show that we can increase the number of programs that are possible to analyze by reducing the amount of computation power that is needed.

- The complexity of some analysis methods depends on both the program size and the number of iterations in the loops. This is the case for *abstract execution*, which we will describe in more detail in **??**. To speed up such analyzes we will remove the assignment statements that do not affect the control flow. Not all variables need to be involved in of loop exit conditions or in branch conditions. Thus the result of assignments of such variables are not needed to determine loop bounds, and their definitions can therefore be removed. When removing such statements we need to consider pointers as well as global variables. The value of a variable may be used to control the program flow when used in one context but not another. Hence it would be possible to remove the corresponding defining statements in latter case but not in the former. To be able to remove as much code as possible we should perform this analysis in a context sensitive manner. The result will be presented in a paper.

- Some language constructs may occur more frequently in industrial real time code than in code from desktop applications. We have an idea that this may concern loop conditions and dependencies between loops. We can find out that if we look into details and use a global view. We aim to inspect code from 5 embedded systems vendors in detail. The result will be presented in a paper.

- The knowledge that we will get from the inspection mentioned above, can be used to introduce suitable syntactical methods to find loop bounds that would else require the user to supply information. Syntactical methods are such that looks for certain patterns or program construct in the program to analyze. It is prepared to act in a certain way when finding such patterns, e.g. make some calculations base on the patterns found, and annotate the loop with the result. Eventually one can find upper loop bounds this way. We will do the syntactical analysis in a global and context sensitive manner. We will consider pointers. If we manage to identify the most frequent loop constructs this way, then we will be able to analyze a larger set of programs than we would have done without this analysis. The result of this work will be presented in a paper.

We will also present the results of the items as listed above as licentiate thesis.

The rest of the paper is organized like this: In Section 2 we will outline the area of WCET analysis with focus on flow analysis. In Section 3 we present some research results that benefit a work in the flow analysis area, as well as its context. Finally, in Section 4 we will explain details about the planned work.

2 Background

The task of calculating an estimated WCET for some code can be divided into three main parts, *flow analysis*, *low level analysis* and *calculation*.

The goal of the flow analysis is to provide the WCET calculation with information about possible upper loop bounds and infeasible paths. Infeasible paths are such paths that can never be executed, because of the conditions. You can see an example in Figure 1 where the path A, B, C, E, F, G can never be executed since the conditions in statements B and E can not be true at the same time.

The information produced by the flow analysis must be safe. This means that one must ensure that the flow constraints, when used by the calculation, will never result in an estimated execution time that is less than the actual WCET. If the flow analysis can not find a single value for the number of iterations of a loop, but rather an interval, then the upper limit of the interval can be use as an safe approximation. One way to do that is to give all the values as a set. Another way is to use a single interval including the lowest possible and the highest possible number of iterations. The latter may result in an overestimated calculated execution time, but it is still safe. We want the overestimation to be as small as possible, but sometimes we need to make compromises in order to make the analysis more efficient.

The low level analysis mainly concerns effects on the execution time from memory hierarchies and pipelines.

With the flow constraints, pipeline and cache effects available, the actual WCET can be calculated by use of the object code and a timing model for the processor in use [9]. The timing model can be more or less complex, e.g., if desired it can include effects of pipelining and caches.

This work will focus on the flow analysis part.

```
    foo(x):
A:   loop(i=1..100)
B:     if (x > 5) then
C:       x = x*2
       else
D:       x = x+2
       end
E:     if (x < 0) then
F:       b[i] = a[i];
       end
G:     bar (i)
     end loop
```

Figure 1: Infeasible path. The path A, B, C, E, F, G can never be executed.

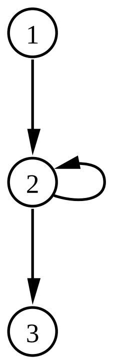

Figure 3: A CFG for a simple loop.

2.1 Flow analysis in WCET

A major issue for the flow analysis is to identify loops and find the upper loop bounds. In order to do this we need a suitable representation of the program code. We use a control flow graph, *CFG*, for that purpose.

```
int fac(int n)
{
   int x;
   x = 1;
   do {
      x *= n;
      n--;
   } while (n > 1);
   return x;
}
```

Figure 2: A simple loop.

2.1.1 Using the control flow graph to identify loops

In Figure 2 you can see a piece of C-code that constitutes a loop. Figure 3 shows the corresponding CFG. If there exist a single node which will always be executed before all other nodes in the loop then this node is called the loop *header*, or entry node (node number 2 in Figure 2), and the loop is said to be a *structured loop*, *natural loop* or *reducible loop.* A loop with more than one possibility to enter the loop body is named *unstructured loop* or *irreducible loop.* Unstructured loops should not be confused with unstructured coding in general. Using control transfer statements like the `goto` statement in C will in most cases not introduce paths in the CFG that differs from those originating from structured constructs in a high level language. An unstructured loop can informally be seen as built from a structured loop with one or more additional jump to some node that is not the header node of the structured loop.

In general, a CFG can contain edges between any two nodes. There can be nested loops, possibly with unstructured and structured loops mixed, also possibly with multiple exits. To identify distinct loops is therefore a non-trivial task.

Unstructured loops may be harmful for some other parts of the flow analysis [33] as well as for the actual calculation parts.

2.1.2 Context sensitive flow analysis

The upper loop bound, i.e. the upper limit on the possible number of iterations, is necessary to be able to calculate the WCET. A function is usually called with different actual parameter values on different call sites. As a consequence a loop in a function may iterate a different number of times, depending on in which context the enclosing function is called.

To be sure that no underestimation of the WCET will be done, we may give a flow fact that is the union of iterations at the various call sites. Anther possibility is to make the analysis *context sensitive.* This means that we specify flow constraints individually for different contexts. This requires the recipients of the flow constraints (i.e. the low level analysis as well as the WCET calculation part) to be able to handle context sensitive flow constraints.

A context sensitive analysis will take into account the certain conditions of a context when making the analysis. If the analysis is made in a context sensitive manner, it will produce a more precise result to the expense of consuming more resources in terms of computation power and memory. There may be different levels of context sensitivity.

2.1.3 Loop dependencies

A loop that is enclosed within another loop, either directly or indirectly because of a function call, may have different loop bounds in different iterations of the outer loop. In Figure 4. The inner loop will iterate a different number of times because of its dependency on the outer loop counter. But also the outer loop may iterate different number of times when the function `bar` is called in different contexts.

```
void bar(int e)
{
   int i, j;
   for (i=0; i<e; i++) {
      j = i;
      while (j > 0) {
         baz();
         j--;
      }
   }
}
```

Figure 4: Context dependent loops.

2.1.4 Multiple loop exits

Some loops have multiple exits like in Figure 5. The presence of multiple exits is not always obvious on the source level. In the loop in Figure 6 there may be multiple exits, since the compiler will probably translate the boolean binary operator to two different branches.

The presence of multiple exits makes it more complex to find the loop bounds.

```
extern int foo;
for (i=0,j=1; i<100; i++, j+=3) {
   if (j > foo) {
      break;
   }
}
```

Figure 5: A loop with multiple exits.

```
extern int foo;
i = 0;
while (i < 100 && i < foo) {
   i++;
}
```

Figure 6: A loop with implicit multiple exits.

2.1.5 Automatic flow analysis

The problem of finding infeasible paths and loop bounds will highly benefit from information from the programmer about this information, so called manual annotations. These annotations can be given in a number of ways, e.g., as an extension of the programming language or in separate file. The need for manual annotations poses several problems:

- It is a tedious work for the programmer.
- Mistakes are easily done.
- An optimizing compiler may reorganize the code, like moving code into or out from loops, unrolling loops etc, hence introducing changes in the CFG. A loop that is annotated in the source code maybe does not even exist after the compilation process is finished.

Therefore many researchers aim to develop strategies to find the flow constraints automatically. A fully automatic analysis means that the programmer does not need to supply any annotations that describe the program flow, only the code as such is needed. Optionally an automatic analysis can demand the programmer to supply specific flow constraints in case it fails to calculate this. If the automatic analysis e.g., fails to bound a certain loop automatically, it can prompt the programmer for the bounds of that loop.

2.1.6 The code representation to analyze

The flow analysis can be performed on either object code, source code or some intermediate representation of the program.

Using source code has the draw-back that some cooperation with a compiler is needed, hence there will be a binding.

Using object code one may get into problems when building the CFG. One reason can be that there are branch instructions that hold the target address in a variable or register. The problem in such a case is that in order to build the CFG, a value range analysis is needed (which is a kind of flow analysis), but the flow analysis needs the CFG. Another effect of making the analysis on the object code level is a stronger binding between the flow analysis and the hardware platform. Analysis on the object code can in some cases perform better if it uses information about the structure of the source code which may be passed by the compiler.

The advantage of making the flow analysis based on the object code is that it can be more independent from the compiler, and all optimizations have been done. The advantage with analyzing on source code level is that the flow analysis is not bound to the hardware architecture, hence this gives a more modular design. Also the building of the CFG is simpler, often there is already some kind of CFG representation present in the compiler.

2.2 How to obtain flow constraints

There are three main approaches for deriving flow constraints; symbolic execution, abstract interpretation and syntactical analysis.

2.2.1 Symbolic execution

In symbolic execution the current program state, like values of variables, may contain symbolic expressions. Such expressions can be evaluated when needed or when the values of the variables involved in the expression are known. The process can informally be described as an interpretation of the program. The purpose is to use it in the flow analysis to try to find constant values of variables in hope to find loop bounds and infeasible paths.

2.2.2 Abstract interpretation and abstract execution

A program can be analyzed using formal semantics. The meaning of the program is seen as the final state that was reached after executing the program with a certain initial state. If concrete semantics and concrete values are be used, then e.g., operational semantics can be used. While making such an analysis the number of iterations of each loop can be calculated, and thus loop bounds can be found for certain input values. However the problem we have to solve is to find loop bounds for a set of input values. The set of possible values can be huge. Repeating the analysis process for all possible combinations of input values would require far too much computational effort. This would in principle be the same problem as using testing by simulation to get the WCET approximation. The purpose of abstract interpretation is to process all possible input values in a single analysis.

Abstract interpretation can be seen as an extension the operational semantics. The value domain is extended to include *abstract values*. An abstract value can be description of possible actual values. It can also contain other values like *undefined*. The semantics of language constructs like operators and expressions need to be redefined to be able to operate on the abstract values.

The methods for the actual interpretation may vary. Gustafsson [13] describes a method where an abstract value is represented by a single interval. He uses an interpretation algorithm that is able to distinguish individual iterations in loops. When we will use the term *abstract execution* for this implementation of abstract interpretation in the rest of this paper.

A major problem with abstract execution is that the calculation efforts may be considerably large, which may lead to too long analyzing time for certain programs [13, 10]. To reduce the computational requirements *merges* are done. Merge here means to merge two or more abstract states, achieved by performing the abstract execution through different paths, to one single state. Such a merge will be made in a safe way, so the resulting state will always include the worst case. An abstract state contains the abstract values of all the program variables that exists in the current scope.

Specific node types or edge types in the CFG can be used as merge points [16]. A typical point for a merge may be the last statement of a loop. A merge at the end of a loop will merge the state of the current iteration with the state of the previous.

2.2.3 Syntactical analysis

In a simple loop construct like the one in Figure 7 it is quite obvious that the number of iterations is 100. The purpose of syntactical analysis is to find loop constructs in the code that are simple enough to be bounded using some kind of pattern matching for common patterns. In a loop a value may be calculated that affects the number of iterations of some other loop. A simple case is if the iteration variable of an outer loop is used as end value of the counter of an inner loop, like in Figure 8. In this and some other certain cases the inner loop can be statically bounded although the termination condition contains a variable.

```
int fac(void)
{
   int x, n;
   x = 1;
   n = 100;
   while (n > 1) {
      x *= n;
      n--;
   }
   return x;
}
```

Figure 7: A simple loop with a constant upper limit.

```
   for (i=0; i<100; i++) {
      for (j=i; j<100; j++) {
         x += j+i;
      }
   }
```

Figure 8: Nested loops where the number of iterations of the inner loop depends on which iteration of the outer loop it is run.

2.2.4 Combined with abstract execution

In [17, 15] a combination of syntactical analysis with abstract execution is described. The point is to benefit the best from both, see Section 3.8.3 for further discussion.

3 Related work

3.1 Loop dependencies

Healy et al. showed in [21] that it can be possible to obtain a total count of iterations for the inner loop in cases like the one in Figure 8. Their method is to express the number of loop iterations as sums and then convert the sums to closed form. Thus no iteration at all is needed for bounding such loops. The method works for, in principle, any loop nesting level. They also showed that loops with nonunit stride (i.e. a loop counter incremented or decremented by some other value than 1) can be expressed as sums, which yields in an exact loop iteration count. The restriction is that the increment or decrement as well as the loop limit (e.g. the expression that the loop counter is compared to) must be a constant. They use *GPAS*, General-Purpose Algebraic Simplifier, which is a part of the tool *Ctadel*, to assist in their calculations.

3.2 Multiple loop exits

Because also a regular boolean expression, like in Figure 6, implies multiple loop exits, we should not be surprised if these are quite common. The study in [28] does also indicate this.

The Florida group, [20, 19], has shown that iteration counts for loops with multiple exits can be calculated, provided that loop increments are constant, and the loop counters are compared to constant values in the loop termination conditions. They also show algorithms to find bounds for loops that contains unknown variables in some of the termination conditions. They can do this as tight as possible, i.e. without losing any of the information that is known.

When the iteration of loops depend on some unknown input, then the analyzer can calculate symbolic expressions for the loop bounds, and prompt the user for values of the unknown variables. It is more likely that the user knows the values (or ranges) of the inputs, than of the loop ranges. Therefore this approach is more reliable than annotating the actual loops. The Florida group has shown also this in detail.

The algorithm is not able to handle dependencies across more than one nesting level, and it also restricts the inner loop not to be enclosed within a conditional branch which can not be calculated.

3.3 Analysis of simple programs

Stappert and Altenbernd [32] developed a method that needs no user annotations. They calculate the WCET for each basic block as the first step, taking in account pipeline effects. The result from this is used in the flow analysis. The method works like this: Create all permutations of paths. Sort them in descending order with respect to execution times. Use symbolic execution on the paths one by one to find if it is a feasible path. As soon as a contradiction of conditional paths indicates an infeasible path the analysis continues with the next path. This goes on until the first feasible path has been found. The permutations actually do not need to be generated, only execution times of pairs of basic blocks needs to taken care of (the basic blocks for `then` and `else` respectively).

They also developed a tool *PTA*, Program Timing Analyzer. The flow analysis part of the tool is integrated entirely into a compiler. The symbolic execution uses actual values extended with *unknown*, and is therefore quite similar to abstract interpretation. PTA is limited to programs without loops. This may seem as a large limitation, but they claim that many tool-generated tasks have no loops. They tested the tool on a suit of some tasks fulfilling that condition. The results show that the calculated WCET is quite tight. The overestimations that were noticed also include effects from imperfections in the calculation part (not only related to the flow analysis part) see [32].

3.4 Annotations and optimizations

One disadvantage with manual annotations, as mentioned in Section 2.1.5, is that the compiler may reorganize the code quite much due to optimizations. This may be a problem even in the case where the annotations has been derived automatically and on the source code level. This problem is addressed by Kirner and Puschner [24]. They suggest an annotation language and a method to keep the flow constraints consistent during the compilation process.

3.5 Realistic Experiments

To make benchmarks that shows the usefulness of a certain algorithm involved in WCET, one needs to have access to real-time programs actually used. There is a benchmark suit available at C-lab in University of Paderborn. Engblom [6, 7] made a lot of interesting findings regarding specific properties of real-time systems, compared to desktop programs. Also there are some preliminary findings in [28] that show some properties of code for real-time systems.

In a comparison between specInt95 and code for embedded systems, Engblom showed that the differences in programming style between desktop applications and embedded systems are significant. Therefore relying on desktop programs only, is not sufficient when testing tools for embedded systems.

Another related investigation, [5, **?**], showed that it can be possible to perform a WCET calculation of systems calls in a certain real-time operating system, *RTEMS*. They found that the code was quite simple; no nested loops, unstructured code or recursion were found. Some function pointers were used.

As we have seen so far, there are no standardized WCET benchmarks with code typical for real-time systems.

3.6 Program Slicing

A program slice is a subset of a program where the elements are statements. The slice has the same possible values of a certain set of variables at a certain execution point (program statement) as the original program. When executing the program slice it will have exactly the same properties as the original program with respect to the set of variables at the program statement of interest.

The purpose is that it will be easier to analyze the program slice than the entire program, since it is smaller. Program slicing were introduced by Mark Weiser [34, 35]. One possible use of program slicing in flow analysis is to remove statements whose execution does not affect the control flow [29].

3.7 Unstructured loops

There are some methods for identifying loops in code potentially containing unstructured loops, DJ graphs [31] being one.

A general method for solving problems induced by unstructured loops is suggested in [27]. The main idea is to use some algorithm for identifying all kind of loops, of which the DJ graph algorithm is one. Using a context sensitive representation with references to the basic blocks rather than the actual basic blocks themselves, copies of any unstructured loop can be made, one for each header node. In case of using the DJ graph algorithm some of the loop copies may expose new loops (not recognized as a separate loop by the DJ graph algorithm in the first stage), since this copying method works like 'unmessing' a loop. This problem in may require to apply the DJ graph algorithm recursively.

3.8 WCET tools

3.8.1 aiT

There is a commercial tool, *aiT*, that uses abstract interpretation [1]. Their analysis builds the CFG from executable code, using task scope. The tool has so far focused on analyzing code from the airplane industry. To make their analysis the user (i.e., the programmer) usually needs to provide some information, e.g. flow constraints. To reduce the workload for the user they have recently developed an improved annotation language [11].

The tool provides some different loop analysis methods. Since this is a commercial tool we have not access to all details about the algorithms used.

3.8.2 BoundT

Holsti et al. [23] has developed a commercial tool, *BoundT* [2], which so far has been primarily used to analyze code from the space industry. The tool builds the CFG from object code. It is more likely that the analysis will succeed if there are debug information present in the object code. This makes BoundT's analysis, to some extent, dependent on the compiler that was used, despite that its input is pure binaries.

They provide a language for a user to make annotations in a separate file. The language makes it possible for the user to refer to e.g. loops in an descriptive way when specifying the number of iterations. Also variables and function invocations can be bound.

The automatic analysis is limited to *counter-based* loops, i.e. loops where a single variable is incremented with a constant value and the loop exit condition is a constant limit.

Using BoundT under certain conditions showed some unexpected problems, concerning the flow analysis, [26]. One of the problems concerned building the CFG from object code.

3.8.3 Prototype research tools

Ermedahl and Gustafsson [10, 13] show how abstract execution can be used to calculate automatic loop bounds, eliminating the need for the programmer to supply manual annotations. They present a total of three different tools. The first tool is described in [10]. This prototype tool handles a subset of the language C. They claim that their method is able to detect false paths as well as calculating safe loop bounds for nested loops. They present experimental results for a single academic program that shows a decent overestimation. In [13] the correctness of the method is proved.

The method is used also as base for a tool that handles a subset of Real Time Talk, *RTT*, see [14]. RTT is an object oriented language for real time systems. Gustafsson concludes that it is possible to analyze programs of 'reasonable size' without manual annotations using this tool. The result is said to be safe and tight. Infeasible paths can be found and thus excluded from the calculation.

In [17, 15] yet another prototype tool is described. This time the tool works on the intermediate code that is generated from a compiler framework, that supports the full C language. Later the same tool was integrated also with the SUIF compiler framework [30].

Also in this tool the flow analysis uses abstract execution. To reduce the computational efforts, the abstract execution only uses a single interval to represent an abstract value that is not a pointer. For pointer values a set representation is used.

There are some more tools, mainly used for research:

- The Florida group has developed a tool that is integrated into a compiler framework. The tool is capable of handling nested loops and loops with multiple exits without annotations.
- Vienna WCET group's tool, [3], requires the analyzed program to be written in wcetC, a dialect of C that is extends C with annotations for WCET flow constraints. The compiler part of the tool is based on gcc. There are some restriction on the programs to be analyzed.
- pWcet, [25], is a tool that also calculates an execution time that might not be the worst case, but the tool also provide the probability for the time to be the actual WCET.
- Heptane, [22], developed at Irisa. Loops needs to be annotated manually.
- Cinderella, [4], is a tool developed at Prince-

ton University. It requires loop bounds to be entered by the user.

4 Research Description

We intend to extend the currently known syntactical analysis methods [20, 19, 21] to find loop bounds. The purpose of this is to supply the WCET calculation with needed flow information. The syntactical analysis is not assumed to be able to find all loop bounds, but we expect it to reduce the computational efforts of other analysis methods like abstract execution. The syntactical analysis may also be able to produce tighter bounds than other analysis methods. We believe that it is possible that these methods together will be efficient enough to make it possible to analyze real world programs with no other need for user interaction than program input. We will compare the performance with and without the support of syntactical analysis.

As a first move we will inspect the loop properties of a number of real time programs. The primary purpose with this part of the thesis is to identify which program constructs regarding loop conditions that are essential to handle by the syntactical analysis.

4.1 Program reduction

In order to make the syntactical analysis more efficient, we will perform a *program reduction*. We will use program slicing, described in 3.6, to remove all assignment statements that do not affect the control flow.

A number of program slices will be created, each of them can be seen as a subset (statements are elements in the set) to the original program. Such a program slice will be created for a conditional branch statement with respect to all variables included in the branch condition. The reduced program will be the union of all the slices.

The removed statements are not needed in the flow analysis, since we are only interested in loop bounds and infeasible paths. The program reduction will be done in a global and context sensitive manner. Pointers will be handled. Various types of pointer analysis will be used in order to see which is most efficient. Parts of this work has already been done as a master thesis, [29].

We expect the program reduction to speed up the analysis time, both for the syntactical analysis and for other analysis passes like abstract execution. The effects of using program reduction will be studied. The analysis time will measured for the syntactical analysis and for abstract execution with and without program reduction. We will also measure the effects of different pointer analysis methods to find out if the effects of a more precise pointer analysis (which may give a more efficient program reduction, which in turn will reduce the workload for the analysis) will outweigh the cost for such a pointer analysis.

We plan to present the results from these measurements in a paper.

4.2 Code inspection

In order to find out which loop properties that are common in code from real applications, we will survey a number of real time applications. We will use the same methods as outlined above in the description of the syntactical analysis. We plan to present the results from the code inspection in a paper (some preliminary results have been obtained in [28]). Some properties that might be useful to get quantified are listed below. The amount of code that will be inspected, will not be enough for statistical significance. Therefor it might be valuable to do a more qualitative investigation concerning some of the findings in the first round.

4.2.1 Program structure

The program structure can be analyzed by examining the call graph. The depth of a call graph can give hints about how much calculation power that is needed for analyzing the code. Extending the call graph with information about the presence of loops will give more information. Comparing a call graph in DAG form with one in tree form will show to what extent functions are reused. Calculating loop bounds in functions called from different sites in a context sensitive manner will cost more in calculation efforts, but may give better (tighter) loop bounds. Recursive functions should be recorded separately. These may be hard to bound syntactically, but simple cases may be possible to handle in case an inter-procedural analysis is performed.

4.2.2 Loop nesting

The nesting level of loops can be counted both locally (per function) and globally (per task or program). It is important to bound deeply nested loops, since they will have the highest influence on the final WCET. Also other analyzing methods (e.g., abstract execution) may suffer from high computational load when analyzing these.

4.2.3 Loop conditions

Reducible loops have a single entry point (we only consider reducible loops since we assume that irreducible loops are already replaced by their multiple reducible loops counterpart [27]).

There may be an arbitrary number of exits from loops. Those with no exits (infinite loops) may occur in real-time systems. We cannot find the loop bound of such a loop syntactically. However, although we can't analyze them we need to count them to conclude how large portions of a program that needs to be excluded from the analysis.

Loops that contain more than one exit branch have been shown to be analyzable syntactically [20], but are in general harder to calculate the bounds of, both in terms of computational efforts and implementation issues. Thus the number of loop exits is of interest as well as the number of targets of these exit branches.

For each termination condition there will be one or more variables involved (otherwise the loop is either equivalent to an infinite loop or with a non-looping construct). Each such variable needs to be carefully investigated. The following properties of the variable may be of interest:

- The initial value.
- The variable update in the loop.
- The operators involved in the exit condition.

Initial values

We are mostly interested in the initial value at the loop termination condition in which it is used (do loops may be handled a bit different, since the initial execution of the loop body might affect the initial value of some variables). The initial value can either be deduced from a constant, or depend on a variable that is updated in an outer loop or depend on an input value to the code.

Initialization from constants. If the initial value only depends on constants this will simplify the analysis, and makes it more likely that we can find a loop bound. The constant value can be found as an assignment from an expression containing only operands that are constants or that are other variables that recursively depend on constants. It is of interest to record the locations of the constants in terms of function nesting level. If all the constants are not present in the same function as were they are used, a more advanced syntactical analysis is needed (e.g., a global analysis).

Initialization from variables updated in an outer loop. There are certain kinds of nested loops where the loop bound of the inner loop can be calculated even if the number of iterations depend on an outer loop induction variable [15]. Therefore these class of initializations are of a certain interest. The following properties needs to be recorded:

- The loop nesting levels between the use and initialization. For example "triangular loops" can be recognized by a syntactical analysis, and it may be possible to find loop bounds in case there is a loop nesting level of one between the two loops.
- The properties of the source to the initialization are of interest. Gerlek et.al., [12], makes a classification of induction variables that may be useful as basis. The problem of finding the bound for a loop can be expected to vary based on these.
- Is the assignment of the initial value done from an expression containing more than one induction variables in outer loops? If so, it will be a harder case to handle.

Initialization from input values. The initial value can in some cases be deduced from some input value. Input to a real-time system may in general occur in various ways. One would be that the code just use some global variable that appears uninitialized to the analyzer (e.g., the code reads from a labeled input port).

Different analyzable units (e.g., tasks) may need to communicate to each other, and this data will appear as inputs. The actual input data can in such case be deduced to some global entity, e.g., global variables. These variables may occur as initialized to the analyzer.

In case of an analysis local to functions, also the function arguments are input.

There is a special problem in identifying input values. The analyzer need to distinguish between those global variables that are input to the analyzed code and those that are rather constants or induction variables in the context of the use in a loop condition. This can normally not be done by looking at a part of the system code in isolation. In general it can be hard to perform an analysis on a complete system because of the interaction of an operating system.

In our inspection all definitions that depend on global variables will be recorded as inputs. When doing the syntactical analysis the user may supply information about which are input variables and the value of these. Optionally a certain pass just to identify intertask communication might be developed. Finding loop bounds if the initial values depend on input is much the same problem as finding it for constant initial values.

Updating of variables in the loop body

One or more of the variables in a loop termination condition must change their value in the loop, or the loop will never terminate using that condition. A simple form is a loop counter, i.e. a variable whose value will contain the current iteration number while executing the loop. Also other forms of updates are of interest. If the value of the variable forms a series which we can identify, there is a good chance that we can calculate the loop bound. The following properties will be recorded:

- Is the update self-referencing, i.e. is the right hand side an expression that includes the target of the assignment (directly or indirectly) within the loop.
- The operator(s) applied in update(s). If only linear updates are involved a calculation of the loop bounds can be expected to be less complicated.
- The other operand(s). This may be constant (directly or indirectly) or an input dependent variable. In both cases it will probably be easier to find loop bounds than for induction variable dependent initialization values.
- Do the other operand alters its value in the loop? In case it does, it is probably harder to find a resulting loop bound (e.g., if an increment is altered in a conditional statement).
- Is the update statement conditional? If so, it is in general not possible to find the wanted series.

4.2.4 Branch conditions

Branch conditions, like if-statements and switches in C, may also be of interest to the syntactical analysis. Infeasible paths can be found if the reaching definition for the involved variables are calculated in a context sensitive manner. The values of variables involved in conditions therefore needs to be recorded in the same manner as initial values of loop conditions.

4.2.5 Arrays

In case array elements are used in place for simple variables in loop termination conditions this imposes difficulties to the syntactical analysis. Such language elements can take many different syntactical forms. Below are listed some that will be recorded in the first round.

- *Initial value.* A variable in a loop termination condition is an array element (or its value depends on an array element), and this variable is loop invariant. For certain sub-cases we might be able to find loop bounds.
- *Update.* A variable in a loop termination condition is updated using an expression containing an array element (or a variable whose value depends on an array element). The updated variable is an induction variable. The index variable might be an induction variable in an outer loop.
- *The "sentinel problem".* There is some loop termination condition that compares an array element with some other value. The array index is an induction variable in the current loop. This category can be tricky to handle. But since we know that strings as well as sometimes pointer arrays are often traversed this way, it is important to know the number of occurrences of this kind. Maybe we can calculate the loop bounds for some sub-cases.

4.2.6 Pointers

The use of pointers in loop termination conditions imposes problems for the syntactical analysis. Pointers that points to a distinct variable in a certain context might give us some hope. We should distinguish these uses of pointers from other.

4.3 Syntactical analysis

In loop termination conditions there will usually be at least one variable present. For each loop there will usually be at least one variable involved in at least one expression that changes its value in the loop, an induction variable. In the simple case this is a loop counter. Such variables will get initial values prior to the loop, and change their value. Other variables involved in termination conditions as well as induction variables, will be initialized either with constants or will get their value from some other variable. In the latter case we can deduce some origin value by use-def chains.

Use-def chains well give us either some expression with only constants, in which case we can just calculate the value of that expression. If, instead, the initialization expression contains on one or more variables, then these variables are inputs to the given scope. It will depend on the scope we use in our analysis, to which extent we will find constant values. We plan to use a whole program scope (or task scope, in case of code for real-time systems), in order to obtain a maximum number of constant initializations of loop variables.

Assume a a function that contains a loop, and that some input parameter is involved in the termination condition of that loop. Such a loop will iterate a different number of times, if the actual value of the parameter is different at different call sites. A whole program analysis will need to consider the values from all call sites to be able to produce flow information that gives a safe WCET. However this may lead to considerable overestimations. Therefore we plan to make the analysis context sensitive. This means that to handle each call separately and assume that the calculation is able to do these too.

A context sensitive analysis can be costly in both computation as well as in memory requirements. To make the analysis more efficient we will perform the analysis first in a local manner where we, when necessary and possible, express loop bounds as symbolic expressions depending on input. Input in this case *input* means input to the analyzed function, i.e. the parameters and global variables that eventually are needed to express the number of iterations.

4.3.1 Detailed description

The calculation can be done in the following manner.

- A first pass builds the call graph for the analyzed entity, which possibly, but not necessarily, is the main function. This graph is a rooted DAG where each node correspond to a function. It requires all files to be analyzed, one by one. This pass will annotate the functions with with references to call sites within the function.

- In a second pass this DAG is traversed in a bottom-up manner. In each node of the DAG a loop tree will be built. This analysis is local to the function, so eventual function calls will not be expanded.

  The loop tree will be traversed in a bottom-up manner. In each function we need to investigate variables with the following properties:

  - Variables that controls the loops contained in the function
  - Variables that maps to variables in parametric expressions of called functions.

  When tracing the definition chains of these variables backwards we will obtain their values as expressions. Such an expression is either possible to evaluate to a plain constant or it will contain one or more variables. These variables are either formal parameters to the function or global variables. All global variables will be handled as if they are parameters. This means that the list of parameters will be extended with the necessary global variables.

  This way we will get some loops for which we know the loop bounds and some that can only be expressed in parametric form. The analysis will annotate the function with both, but only the parametric ones will need to be available for callers to the current function. The annotation will include a reference to the location of the loop within the tree.

  Called functions will be handled in a similar way. We are only interested in the available annotations of the called functions. When we have applied the information obtained by the use-def analysis we can divide them into two types: those that can be evaluated to constants and those that are still on parametric form. Both types will generate new annotations of the current function. The new annotation will either contain final values or a parametric form of the loop bounds. They will contain references to the the precise call site within the function as well as to the annotation of the called function. This way only those annotations that are needed will be propagated upwards in the call graph. At the same time we will maintain a context sensitive information when needed.

Each annotation may have two representations in parallel of the same information. A summation form is needed in case there will be a dependency to some outer loop's induction variable. There can also be a closed form of the summation. This may speed up the in case that the function is called from many sites and all of them can supply constants to the variables in the parametric form. The closed form will be added on demand to avoid the transformation being done in case never needed.

When reaching the root of the analyzed entity, there may still be formal parameters and global variables whose values are neeeded but unknown. These must be input data to the analyzed code. The user can be informed about which input data that are needed, and will be able to supply these.

The user supplied program input is a possibly empty set of values, and a possibly empty set of ranges. These values will be used to calculate the upper and lower bounds of all parametric loop information that was propagated up to the root node.

- A third pass will calculate flow constraints from the information obtained during the first pass. This pass will be run in a top-down manner, to be able to bring necessary scope information down to the location of the flow fact. Engblom et al has designed a language, see [8], that may be useful to express context sensitive flow constraints.

Annotations containing final information about the final loop bounds will have the form

```
<{v1, v2, ...vn}, {I1, I2, ... In}>
```

where vi are constant values and Ii are intervals with a constant range.

Annotations for parametric expressions will have the form

```
<es, ec>
```

where ec is optional.

es has the form

```
es -> sarith
sarith -> sarith op sarith | variable
       | constant | floor(arith)
       | sum(arith, arith, sarith)
op -> + | - | * | /
```

ec has the form

```
ec -> arith
arith ->arith op arith | variable
       | constant
op -> + | - | * | /
```

4.4 The effects of syntactical analysis

We believe that a syntactical analysis as described above can speed up the analysis considerably for certain programs. We aim to do measurements where we perform an analysis based on abstract execution. We will compare the effects of adding syntactical analysis as a preprocessing step. A context sensitive syntactical analysis will require more computations and memory than with an insensitive analysis. Since we beleave that the the result will be more precise. Compared to the abstract execution it will still be fast. We will put efforts in finding useful benchmarks for the comparisions, and write a paper to make our results available for other researchers.

In some cases it may be possible to calculate final values of all the flow controlling variables that are updated in the loop body. If we can do that, the loop will be annotated with these values. Other analyzes may benefit these in order to speed up their calculation time. The effect of calculating final values of such variables will be measured by comparing analysis with and without this syntactical analysis. The method can also be validated by comparing the requirements that we have on the source code with those of other current research methods.

We plan to present the results from these measurements in a paper.

4.5 Written papers

- Elimination of Unstructured Loops in Flow Analysis, [27]
- Inspection of Industrial Code for Syntactical Loop Analysis, [28]
- A Tool for Automatic Flow Analysis of C-programs for WCET Calculation, [18]
- A Prototype Tool for Flow Analysis of C Programs, [17]

4.6 Planned papers

- Speed up of WCET analysis by program reduction, EMSOFT or RTCSA
- Loop properties of industrial real-time programs, submitted during spring 2006
- Speed up of WCET calculation by syntactical analysis, submitted during spring 2006

4.7 Courses

- Program Language Semantics, 5p
- Processoriented programming, 5p
- Logic Programming, 5p
- Research Methodology, 5p
- Algorithm Analysis, 5p
- Functional Programming, 5p (not finished)
- Program Analysis, 5p (not started)

4.8 Project plan

Spring 2005

- Course in program analysis.
- Extending the program reduction with pointer analysis, and making measurements as described above.

Spring 2006

- Make code inspection of loop properties as described above. This includes implementations.
- Implement syntactical analysis as described above and make measurements. This includes implementations, but it should be possible to re-use quite a lot from the code inspection.
- Writing the licentiate thesis, this will be based on a the following papers:
  - Elimination of Unstructured Loops in Flow Analysis, [27]. One possibility is to improve this paper a bit and refere to it as a technical report.
  - Speed up of WCET analysis by program reduction, possibly on RTCSA, or other conference during spring 2005
  - Loop properties of industrial real-time programs, submitted during spring 2006
  - Speed up of WCET calculation by syntactical analysis, submitted during spring 2006

References

[1] ait: Worst case execution time analyzers, Aug 2004. URL: `http://www.abstract-interpretation.com/ait/`.

[2] Bound-t execution time analyzer, Aug 2004. URL: `http://www.tidorum.fi/bound-t/`.

[3] calc wcet 167, Aug 2004. URL: `http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/`.

[4] Homepage for the `cinderella` system, Aug 1997. URL:`http://www.ee.princeton.edu/~yauli/cinderella-3.0/`.

[5] A. Colin and I. Puaut. Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System. Technical Report Publ. Interne No 1277, IRISA, Nov 1999.

[6] J. Engblom. Static Properties of Embedded Real-Time Programs, and Their Implications for Worst-Case Execution Time Analysis. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*. IEEE Computer Society Press, Jun 1999.

[7] J. Engblom. Why SpecInt95 Should Not Be Used to Benchmark Embedded Systems Tools. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, May 1999.

[8] J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proc. 21th IEEE Real-Time Systems Symposium (RTSS'00)*, Nov 2000.

[9] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis.* PhD thesis, Uppsala University, Dept. of Information Technology, Box 325, Uppsala, Sweden, Jun 2003. ISBN 91-554-5671-5.

[10] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer Verlag, Aug 1997.

[11] C. Ferdinand, R. Heckmann, and H. Theiling. Convenient User Annotations for a WCET Tool. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, 2003.

[12] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables, detecting and classifying sequences using a demand-driven ssa form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, Jan 1995.

[13] J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation.* PhD thesis, Department of Computer Systems, Information Technology, Uppsala University, May 2000.

[14] J. Gustafsson. A Prototype Tool for Flow Analysis of Object-Oriented Programs. In *The 5th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2002)*, April 2002.

[15] J. Gustafsson, N. Bermudo, and L. Sjöberg. Flow Analysis for WCET calculation. Technical Report 0547, ASTEC Competence Center, Uppsala University, URL: `http://www.mrtc.mdh.se/publications/0547.ps`, Mar. 2003.

[16] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a Flow Analysis for Embedded System C Programs . In *8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005)*, Feb 2005.

[17] J. Gustafsson, B. Lisper, N. Bernmudo, C. Sandberg, and L. Sjöberg. A Prototype Tool for Flow Analysis of C Programs. In *WCET 2002 Workshop Wienna*, pages 9–12, aug 2002.

[18] J. Gustafsson, B. Lisper, C. Sandberg, and N. Bermudo. A Tool for Automatic Flow Analysis of C-programs for WCET Calculation. In *8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, Jan 2003.

[19] C. Healy. *Automatic Utilization of Constraints for Timing Analysis.* PhD thesis, Florida State University, August 1999.

[20] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding Loop Iterations for Timing Analysis. In *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, Jun 1998.

[21] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, May 2000.

[22] Homepage for the `heptane` system, Mar 2003.
URL: `http://www.irisa.fr/aces/work/heptane-demo/heptane.html`.

[23] N. Holsti, T. Långbacka, and S. Saarinen. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, Sep 2000.

[24] R. Kirner and P. Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*. IEEE Computer Society Press, Jun 2001.

[25] Probabilistic worst case execution time analysis, Aug 2004.
URL: `http://www.pwcet.com/`.

[26] M. Rodriguez, N. Silva, J. Estives, L. Henriques, and D. Costa. Challenges in Calculating the WCET of a Complex On-board Satellite Application. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, 2003.

[27] C. Sandberg. Elimination of Unstructured Loops in Flow Analysis. In *WCET 2003 Workshop Porto*, 2003.

[28] C. Sandberg. Inspection of Industrial Code for Syntactical Loop Analysis. In *WCET 2004 Workshop Porto*, 2004.

[29] L. Sjöberg. Elimination of variables that do not affect control flow in C-programs. Technical Report TR0216, Mälardalen University, URL: `http://www.idt.mdh.se/utbildning /exjobb/files/TR0216.ps`, June 2002.

[30] R. Söderström. TONIC - Integrating an existing compiler framework with WCET Flow Analysis and Calculation. Technical Report TR0275, Mälardalen University, URL: `http://www.idt.mdh.se/utbildning /exjobb/files/TR0275.ps`, June 2003.

[31] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, 1996.

[32] F. Stappert and P. Altenbernd. Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[33] E. Vivancos, C. Healy, M. Mueller, and D. Whalley. Parametric Timing Analysis. *ACM SIGPLAN Notices*, 36(8):88–93, Aug. 2001.

[34] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[35] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.