

The Programming Language WCETC *

Raimund Kirner
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
A-1040 Wien, Austria
raimund@vmars.tuwien.ac.at
FAX +43(0)1 58 69149

Revision 0.5.0, September 20, 2001

Abstract

Static worst-case execution time analysis of programs requires the knowledge of runtime information that is not a priori defined by the sum of all functional program statements. This information depends on the dynamic assignment of input variables. The runtime behaviour can be modeled when having some knowledge about the dynamic assignments of input variables. The presented programming language WCETC allows to express this information directly inside the source code. This enables the calculation of tight WCET bounds by processing this information directly by the compiler.

1 Introduction

This report defines the programming language WCETC. This language is especially designed to allow the static calculation of the worst-case execution time (WCET).

The language WCETC is derived from ANSI C with grammar extensions to express runtime information inside the source code. Additional restrictions are made to enforce a statically predictable program behaviour.

*This work has been supported by the IST research project "Systems Engineering for Time-Triggered Architectures (SETTA)" under contract IST-10043.

A prototype implementation of a WCET analysis tool was developed and described in [1]. A survey on how the WCET calculation is done internally can be found in [4]. This approach also supports code optimizations performed by the compiler during optimizations. Experiments showing the effects of using different compiler optimizations are documented in [2].

This report is structured as follows: Section 2 introduces the grammar extensions for timing annotations added to ANSI C. The required restrictions on the language to perform static WCET analysis are listed in Section 3. Additional features to receive more results on WCET analysis for advanced applications are described in Section 4. Section 5 concludes the presented features of the programming language. A header file to support portable programming for both ANSI C and `wcetc` is listed in Annex A. Annex B shows an example program written in `wcetc` (`wcetc` is a compatibility version of `wcetc` as described in Section 2.5).

2 Basic grammar of the language `wcetc`

This section lists those parts of the whole C grammar which are necessary to explain the annotations for WCET analysis. The syntax of the grammar is given in an EBNF-like (extended Backus-Naur form) annotation. The symbol `$` is used to express productions to an empty token. Some native C-nonterminals are defined informally in the form of comments (e.g. `INT_CST`, `EXPR`) to remain focus on the grammar extensions for WCET analysis.

```

INT_CST: /* Expression, formed by integer constants, incomplete */
        INT_CST "+" INT_CST |
        INT_CST "-" INT_CST |
        INT_CST "*" INT_CST |
        INT_CST "/" INT_CST |
        INT_CST "%" INT_CST |
        "(" INT_CST ")" |
        NUMBER
        ...

EXPR: /* Similar to INT_CST, but much more general (variable) */

XEXPR: EXPR | $

IDENTIFIER: /* standard identifier in ANSI C */
C_STMT: /* standard C statement */

```

COMPARE: "<" | "<=" | "==" | ">=" | ">"

LOOP: DO_LOOP | WHILE_LOOP | FOR_LOOP

CODE_INFO: BLOCK_INFO | BUILD_INFO

2.1 Constants for WCET Cycles/Iterations

The specification of constants for iteration counts or cpu-cycles is an extended version of the grammar INST_CST from ANSI C. It allows the use of additional built-in functions to model the control flow paths.

```
CYC_CST: "pow" "(" CYC_CST "," CYC_CST ")" |  
        "min" "(" CYC_CST "," CYC_CST ")" |  
        "max" "(" CYC_CST "," CYC_CST ")" |  
        "log" "(" CYC_CST ")" |  
        "log2" "(" CYC_CST ")" |  
        "log10" "(" CYC_CST ")" |  
        CYC_CST "^" CYC_CST |  
        INT_CST
```

The mathematical meaning of the additional tokens is the following:

pow(x,y) ... calculates x^y .

min(x,y) ... selects the minimum of x and y .

max(x,y) ... selects the maximum of x and y .

log(x) ... calculates $\ln(x)$ (logarithm with the Euler Number as base).

log2(x) ... calculates $\frac{\ln(x)}{\ln(2)}$.

log10(x) ... calculates $\frac{\ln(x)}{\ln(10)}$.

2.2 Support for loop bounds

2.2.1 do-Loop

This is the standard do-while loop of ANSI C extended with a loop bound annotation in the header.

```

DO_LOOP: "do"
         "maximum" CYC_CST "iterations"
         C_STMT
         "while" "(" EXPR ")" ";"

```

2.2.2 while-Loop

This is the standard while loop of ANSI C extended with a loop bound annotation in the header.

```

WHILE_LOOP: "while" "(" EXPR ")"
           "maximum" CYC_CST "iterations"
           C_STMT

```

2.2.3 for-Loop

This is the standard for-loop of ANSI C extended with an loop bound annotation in the header.

```

FOR_LOOP: "for" "(" XEXPR ";" XEXPR ";" XEXPR ")"
         "maximum" CYC_CST "iterations"
         C_STMT

```

2.3 Generic control flow restrictions

2.3.1 Marker and Scopes

This is the general construct to express restrictions in the runtime behavior of programs. Also the bounds for all the loops described above are internally transformed to such constructs. The marker is just to label a path in the flow graph. These path names are then used to express restrictions for the execution count of this path at the end of each WCET scope. The semantic of restrictions is to declare the maximum execution count of certain paths in the flow graph (named by a marker) in relation to the execution count of the start node of this scope. Therefore it is for example possible to surround two sequential loops with a scope and declare the bound of both loops dependent from the other one. The latter concept was also introduced as *"loop-sequences"* in the literature.

The productions of the C_STMT are extended by the SCOPE statement, which is like the standard block statement given by curly brackets.

```

C_STMT: __standard_C_stmt__ | SCOPE

```

```

MARKER: "marker" IDENTIFIER ";"

MC_STMT: MARKER | C_STMT
MC_STMTS: MC_STMT MC_STMTS | $

SCOPE: "scope" IDENTIFIER "{"
      MC_STMTS
      RESTRICTIONS
      "}"

RESTRICTIONS: RESTRICTION RESTRICTIONS | $

RESTRICTION: "restriction" MARKER_MULT IDENTIFIER
            RESTRICTION_LIST RESTRICTION_LIMIT ";"

MARKER_MULT: "(" INT_CST ")" "*" | $

RESTRICTION_LIST: "+" MARKER_MULT IDENTIFIER RESTRICTION_LIST | $

RESTRICTION_LIMIT: COMPARE CYC_CST |
                  COMPARE MARKER_MULT IDENTIFIER RESTRICTION_LIST

```

The declared markers are not bound to a certain scope. They can be used for restrictions in any declared scope enclosing them. The following annotated C code shows the possibilities of this annotated language to express restrictions in the execution frequency:

```

void modify(int arr[]) {
    int i;
    scope loop1 {
        for (i=0; i<50; i++) wcet_maximum 50 wcet_iterations {
            if (arr[i]<5) {
                marker case1;
                modify_1(arr[i]);
            }
            if (arr[i]>3) {
                marker case2;
                modify_2(arr[i]);
            }
        }
    }
}

```

```

        else {
            marker case3;
            modify_3(arr[i]);
        }
    }
    restriction case1 <= (5);          /* Only 10% match condition. */
    restriction (3)*case2 <= case3; /* arr[i]<=3 is more likely. */
} /* end of scope */
}

```

This example uses loop bounds as main restriction of the execution time. But there are also sharper restrictions added by the use of scope/marker. The marker *case1* is used to express a restriction relative to the execution number of the current scope. The markers *case2* and *case3* are used to express a restriction of two alternative execution paths inside a scope.

2.4 Specifying additional execution time

To make WCET analysis more flexible, it is also possible to add manually execution time to a certain program flow path.

The productions of the C_STMT are extended by the ADD_CYCLES statement, which looks similar to a normal function call.

```

C_STMT: __standard_C_stmt__ | ADD_CYCLES
ADD_CYCLES: "addcycles" "(" CYC_CST ")" ";"

```

With the above statement cpu cycles can be added to a specific program flow path. This additional time is added to the time of the basic block where is this statement is embedded. Changing the execution time of a basic block can be used to analyze the WCET behaviour for code optimization. It can also be used, when the execution time of a function call is unknown.

2.5 Compatibility Version WCETCc

The programming language WCETC introduces several new keywords to ANSI C. When compiling existing programs it could occur that these keywords are used as normal identifiers, which leads to compiling errors. Therefore, an adapted version of the programming language will be defined that tries to avoid this problem. This is done by adding the prefix "wcet_" to every keyword introduced by WCETC. This compatibility version of the programming language will be called WCETCc.

A listing of the different keywords of WCETC and WCETCc is given by the following table:

Keyword in WCETC	Keyword in WCETCc
maximum	wcet_maximum
scope	wcet_scope
iterations	wcet_iterations
marker	wcet_marker
restriction	wcet_restriction
addcycles	wcet_addcycles

3 Restrictions of the language

The general syntax of the language WCETC is based on the syntax of ANSI C. The differences are on the one side the added annotations to guide the compiler for WCET analysis and on the other side some restrictions, to make the behavior of the program more predictable for static analysis. The following restrictions are made to enable static WCET analysis:

- No use of function-pointers. This is because the use of function-pointers is only useful if the called function should be changed at runtime. But this makes it impossible for static analysis without complex contextual methods.
- No recursive function calls are allowed (neither direct nor indirect). For direct recursion it would be possible to extend the programming language with additional annotations to describe the maximum recursive call-depth. But this concept is not so well suited for indirect recursive function calls.
- No use of *goto*. Because this construct destroys the well-formed loop structures and therefore dismisses the meaning of loop bounds.
- No use of *setjmp()/longjmp()*. These constructs are designed for low-level error handling. But this is in fact like *goto* and destroys the well-structured flow graph.
- No use of *signal()*. This UNIX specific mechanism does not make any sense in our context, because signal handling is asynchronous. It is therefore not possible to analyze this statically.
- No use of *exit()*. This gives additional paths for the whole flow graph to the end of the program and prevents a structured hierarchical analysis.
- Take care on using library functions. The tool is designed to use a description file for the WCET of functions, where the source code is not given or analysed by the tool. But this possibility would be a threat to undervalue the WCET of this function. This could also be the case due to a replacement of library functions by other implementations.

These are not restrictions to prevent raised exceptions at runtime due to incorrect operations like invalid data pointers. So the calculated maximum execution time of course relies on the correct behavior of the program at runtime.

All these given restrictions have been proposed in [5]. They have defined a real sub-set of ANIS C.

4 Extensions for code generation tools

The programming language WCETC was designed to be included into a WCET analysis framework that allows the modeling of an application by Matlab/Simulink with the capability of automatic code generation. A compiler with integrated WCET analysis functionality compiles the generated C code and performs WCET analysis.

To map the results from the analysis back to the simulation model, it is required to have the possibility to express some information about the simulation model inside the C code. Several new keywords have been introduced to enable this. These new keywords are summarized by the following table:

Keywords for both WCETC & WCETCc
wcet_buildinfo
wcet_blockbegin
wcet_blockend

These keywords are described in more detail in the following subsections.

4.1 Extension to the Grammar

4.1.1 Block-Info

These statements indicate the beginning and end of the corresponding block in a Matlab/Simulink model. These statements cannot be hierarchically nested. The first parameter describes the name of the current Matlab/Simulink block. The second parameter of `BLOCK_START` indicates whether the WCET of the statements inside this block has to be calculated or it is specified as a known constant. A value of -1 means, the WCET has to be calculated. For a value ≥ 0 the WCET of this Block is taken as being the given constant value.

```
BLOCK_INFO: BLOCK_START | BLOCK_STOP
```

```
BLOCK_START: "wcet_blockbegin" "(" STRING_CST "," CYC_CST ")" ";"
```

```
BLOCK_STOP: "wcet_blockend" "(" STRING_CST ")" ";"
```

4.1.2 Build-Info

This statement gives information about the build version of the C source code from the Matlab/Simulink model. The argument is a string constant, containing information about the build process.

```
BUILD_INFO: "wcet_buildinfo" "(" STRING_CST ")" ";"
```

This information is not required for WCET analysis, just only for version management.

4.2 Integration of WCET Analysis into a Simulation Environment

The keywords introduced in this section are used to express additional information about the application into the C code and to get more detailed results from the WCET analysis.

An example for the integration of WCET analysis based on C code into Matlab/Simulink to enable WCET analysis at higher abstraction level is described in [3].

5 Summary and Conclusion

WCET analysis requires additional knowledge about the runtime behaviour. The programming language WCETC was designed to express these required information directly inside the source code. WCETC is derived from ANSI C but includes additional constructs to express information about loop bounds or infeasible/feasible paths. This information will be used by a WCET analysis tool to determine the runtime behaviour and calculate the WCET. WCETC has also some restrictions to ANSI C to make the required information about the runtime behaviour fully known at compilation time.

In addition, WCETC contains support for additional WCET information. This was clarified by describing a concrete application build on top of WCETC.

References

- [1] R. Kirner. Integration of Static Runtime Analysis and Program Compilation. Master's thesis, Technische Universität Wien, Vienna, Austria, May 2000.

- [2] R. Kirner and P. Puschner. Consideration of Optimizing Compilers in the Context of WCET Analysis. In *Informatiktage 2000, Fachwissenschaftlicher Informatik-Kongreß*, pages 123–126, Bad Schussenried, Germany, October 2000. GI Gesellschaft für Informatik e.V.
- [3] R. Kirner and P. Puschner. Integrating WCET Analysis into a Matlab/Simulink Simulation Model. In *In Proceedings of the 16th IFAC Workshop on Distributed Computer Control Systems*, Sydney, Australia, November 2000. School of Computer Science and Engineering, UNSW.
- [4] R. Kirner and P. Puschner. Supporting Control-Flow-Dependent Execution Times on WCET Calculation. In *Deutschsprachige WCET-Tagung*, Paderborn, Germany, October 2000. C-Lab.
- [5] The Motor Industry Software Reliability Association MISRA. *Guidelines For The Use Of The C Language In Vehicle Based Software*. ISBN 0-9524156-9-0. MIRA, April 1998.

A Portable Usage of wcetC

To achieve portability when writing in WCETC/WCETCc code with standard C code, the include file `wcet.h` is provided.

This include file defines several macros for WCET annotations. The text replacements by these macros is controlled by extra definitions. The definition of `LANG_WCET` before including the header (`#include "wcet.h"`) controls the kind of WCET annotations. If `LANG_WCET` is not defined, then the default value of `WCETC` is used. When `LANG_WCET` is defined, then the resulting source language for the WCET annotations is given in the following table:

Definition	Source Language for Annotations
<code>#define LANG_WCET WCETC</code>	WCETC
<code>#define LANG_WCET WCETCC</code>	WCETCc
<code>#define LANG_WCET <other value></code>	ANSI C

The rest of this section is a printout of the file `wcet.h`:

```

/*****
 *
 * Header file for the use of wcetC/wcetCc, a
 * language to enable static WCET analysis
 * on standard ANSI C.
 * (wcetCc is like wcetC, but with the prefix
 * 'wcet_' before each wcet key word.
 *
 *
 * (c) 2001 Raimund Kirner, TU-Wien,
 *         raimund@vmars.tuwien.ac.at
 *
 *****/

/* Constants for LANG_WCET to select the right
 * keywords.

#define OTHER    0x00 /* no annotations */
#define WCETC   0x01 /* wcetC
#define WCETCC  0x02 /* wcetCc

/* If LANG_WCET is not defined, the default
 * language is always WCETC

```

```

#ifndef LANG_WCET
#define LANG_WCET WCETC
#endif

/* Definitions of wrappers for the wcetC grammar */

#if defined(LANG_WCET) && ((LANG_WCET)==WCETC || (LANG_WCET)==WCETCC)
#if (LANG_WCET) == WCETC /* WCETC */
#define WCET_LOOP_BOUND(x)    maximum (x) iterations
#define WCET_SCOPE(x)        scope x
#define WCET_MARKER(x)        marker x
#define WCET_RESTRICTION(x)   restriction x
#define WCET_ADD_CYCLES(x)    addcycles(x)
#else /* WCETCC */
#define WCET_LOOP_BOUND(x)    wcet_maximum (x) wcet_iterations
#define WCET_SCOPE(x)         wcet_scope x
#define WCET_MARKER(x)        wcet_marker x
#define WCET_RESTRICTION(x)   wcet_restriction x
#define WCET_ADD_CYCLES(x)    wcet_addcycles(x)
#endif
#define WCET_BLOCK_BEGIN(x,y) wcet_blockbegin(x,y)
#define WCET_BLOCK_END(x)     wcet_blockend(x)
#define WCET_BUILD_INFO(x)    wcet_buildinfo(x)
#else /* ANSI C */
#define WCET_LOOP_BOUND(x)
#define WCET_SCOPE(x)
#define WCET_MARKER(x)
#define WCET_RESTRICTION(x)
#define WCET_ADD_CYCLES(x)
#define WCET_BLOCK_BEGIN(x,y)
#define WCET_BLOCK_END(x)
#define WCET_BUILD_INFO(x)
#endif

```

B Sample Application for wcetC

The following code shows the usage of annotations when programming in WCETC. The code uses the macro definitions described in Annex A.

```
#ifndef LANG_WCET
#define LANG_WCET WCETCC
#endif
#include "wcet.h"

#define N_EL 10

/* Sort an array of 10 elements with bubble-sort */
void bubble (int arr[])
{
    /* Definition of local variables */
    int i, j, temp;

    /* Main body */
    WCET_SCOPE(BS)
    {
        for (i=N_EL-1;
            i > 0;
            i--)
            WCET_LOOP_BOUND (N_EL - 1)
            {
                for (j = 0;
                    j < i;
                    j++)
                    WCET_LOOP_BOUND (N_EL - 1)
                    {
                        WCET_MARKER (M);
                        if (arr[j] > arr[j+1])
                        {
                            temp = arr[j];
                            arr[j] = arr[j+1];
                            arr[j+1] = temp;
                        }
                    }
            }
        WCET_RESTRICTION (M <= (N_EL*(N_EL-1)/2));
    }
}
```

The code represents an implementation of the bubble sort algorithm. The loop bounds are given by the maximum length of the vector to be sorted.