Calculating the maximum execution time of real-time programs

Article *in* Real-Time Systems · January 1989

DOI: 10.1007/BF00571421 · Source: DBLP

2 authors, including:

Peter P. Puschner
TU Wien
172 PUBLICATIONS **5,849** CITATIONS

Calculating the Maximum Execution Time of Real-Time Programs

P. Puschner*, Ch. Koza
Institut für Technische Informatik
Technische Universität Wien
peter@vmars.uucp, koza@vmars.uucp

Version 1; April 7, 1989

Abstract

In real-time systems the timing behaviour is an important property of each task. It has to be guaranteed that the execution of a task does not take longer than the specified amount of time. Thus, a knowledge about the maximum execution time of programs is of utmost importance.

This paper discusses the problems for the calculation of the maximum execution time (MAXT ...MAximum eXecution Time). It shows the preconditions which have to be met before the MAXT of a task can be calculated. Rules for the MAXT calculation are described. Triggered by the observation that in most cases the calculated MAXT far exceeds the actual execution time, new language constructs are introduced. These constructs allow programmers to put into their programs more information about the behaviour of the algorithms implemented and help to improve the self checking property of programs. As a consequence, the quality of MAXT calculations is improved significantly. In a realistic example, an improvement factor of 11 has been achieved.

*This work has been supported by Digital Equipment Corporation under contract EERP/AU-011

1 Introduction

The significant difference between real-time systems and other computer systems is the importance of correct timing behaviour. Each hard real-time task has a deadline that has to be met, otherwise the real-time system fails. As a consequence in a real-time system it has to be guaranteed that each task finishes before its deadline, even in worst case, i.e. when the task's execution takes a maximum amount of time.

Obviously the worst case execution time of a task — we call it maximum execution time — is of significant importance for the construction and verification of real-time systems.

In many articles about scheduling in real-time systems the maximum execution times of tasks are assumed to be known. One example is the classic article about scheduling in hard real-time systems by Liu and Layland [Liu 73]. They assume that the run-time for each task is constant, i.e. that it does not exceed a known amount of time. In [Mok 84] the properties and impacts of the use of semaphores, rendezvous constructs, and monitors on real-time systems are discussed. Mok's work is also based on the assumption that the maximum execution time of program blocks is known.

Kligerman and Stoyenko [Klig 86], [Stoy 87] address the problem of a worst case analysis of tasks' run-time properties. They discuss the real-time programming language *Real-Time Euclid.* Real-Time Euclid is defined in a way that allows the calculation of the maximum execution time for every program. We will come back to some of the language's concepts later.

In Leinbaugh's papers [Lein 80], [Lein 82], and [Lein 86] one can find discussions on guaranteeing response times in hard real-time systems. Leinbaugh takes into account task priorities, mutual exclusion, resource conflicts, task communications, and interrupt handling. The MAXT of tasks is expected to be known.

It is the main focus of the MARS (MAintainable Real-time System) research group at the Technical University in Vienna to build a distributed real-time system with a deterministic, guaranteed timing behaviour [Kope 89]. The design system of MARS [Senf 89] integrates all steps from system design to programming in the small. Tasks are scheduled statically by a pre run-time scheduler. The scheduler takes into account the precedence constraints describing synchronization needs and dependencies among tasks, the maximum execution time of tasks, and their activation frequencies and produces a dispatch table which is interpreted at run-time [Fohl 88]. The maximum execution time is provided by a special tool based on an analysis of task source codes.

In this paper the different aspects of the maximum execution time calculation of real-time programs are discussed.

First we will show that the maximum execution time cannot be calculated for an arbitrary program. Problems for the MAXT calculation are described. This leads to some restrictions for analyzable programs and to the introduction of bounded loops.

In the following a description of some simple rules for the calculation of the maximum execution time of programs which obey the restrictions is presented. Discussing the quality of the results of the MAXT calculation, it will be seen that it is necessary to introduce new programming language constructs which provide a means to state more information about the application context of programs in order to be able to compute a bound for the maximum execution time which is much closer to the real maximum execution time. Language constructs called markers, scopes, and loop sequences are described.

2 Preconditions for the Maximum Execution Time Calculation

In this article we analyze the software aspects of the calculation of the maximum execution time of programs. We make the assumption that the behaviour of the underlying hardware and operating system is deterministic and known (this framework is provided by our MARS system [Kope 89]). This implies that the timing behaviour of all hardware components and the effects of caching, pipelining and DMA on task performance are predictable. On the other hand the operating system must provide static memory management (no paging with statistical behaviour), system calls with a calculable timing behaviour, and the absence of asynchronous interrupts. Task synchronization is provided by the pre run-time scheduler and thus does not produce any overhead at run time.

We define the technical terms application specific maximum execution time and calculated maximum execution time as follows:

Definition 1 (Application Specific Maximum Execution Time ... $\mathbf{MAXT_A}$) *The Application Specific Maximum Execution Time of a program is the time it maximally takes to execute this program in the given application context, provided that all needed resources are available, the program is not interrupted and the performance of the hardware is known.*

Note that the application specific maximum execution time of a task is the maximum CPU time that the task can actually consume. Trying to get a value for the timing behaviour of a task by an analysis of its source code — this is what can be done by a software tool — one can often derive only a high upper bound for the maximal time consumption of a task. This is due to the fact that the program code does not contain the full information about the application context of a task (for more details see section 4). Hence, we define the term of the calculated maximum execution time.

Definition 2 (Calculated Maximum Execution Time ... $\mathbf{MAXT_C}$) *The Calculated Maximum Execution Time of a task is the least upper bound for the $MAXT_A$ of this task that can be derived from the task's program code.*

In order to calculate the maximum execution time of a task the MAXT (for better readability we will use this abbreviation instead of $\mathrm{MAXT_C}$ in the rest of the paper) of all parts of that task — sequences, loops, etc. — must be computable. This suggests that full information about the control flow and constraints for the control flow in the worst case, i.e. when every program part executes as long as possible, have to be known for all language constructs of the programming language used.

The main problem is that the control flow of a program at run-time depends on the input data and the current variable settings. The values of variables used in conditions (loop conditions or conditions of alternatives) and the values of pointers to functions determine the control flow and as a consequence the timing properties of each task. Since it is impossible to simulate the execution of a task for all its possible variable settings and to determine if the task terminates or how long it takes to execute (termination problem), some restrictions have to be made in order to get analyzable programs.

The problems in detail are:

- In most current programming languages the programmer does not have to declare the maximum number of iterations or a time limit for the *loops* he programs. This imposes a problem on the MAXT calculation, because the maximum iteration number or a time limit for the loop execution generally cannot — or only with great effort — be extracted from the loop condition. Thus the time maximally spent in a loop cannot be calculated in most cases.
- The usage of *recursions* leads to a similar problem. The maximal depth of recursive procedure calls, and consequently the MAXT for recursions, cannot

be determined statically, since these attributes depend on the variable state at program execution time.

A related problem of using recursions in real-time applications is that the demand of stack space cannot be determined before run-time. Therefore, the maximal amount of stack space needed at run-time cannot be allocated statically as is done in many real-time systems.

- *Parameters and pointers to functions* can reference functions of distinct timing properties. Because of the dependence of the MAXT on the different functions it does not make sense to assign the maximum time it takes to evaluate a function to the MAXT for a function referenced.

  Furthermore the use of pointers to functions provides a means for the implementation of recursions (see above for more details).

- The danger of *goto* usage is that one can write programs which lack any structure in the sense of structured programming. These programs cannot be analyzed by an automated software analysis tool.

We make the following restrictions in order to eliminate the problems listed above (some are mentioned in [Klig 86]):

- Programs must not contain any (direct or indirect) recursions. Recursive algorithms have to be either replaced by iterative ones or transformed into non recursive schemes by applying program transformation rules [Darl 78].

- The absence of function variables and parameters is enforced in programs that the MAXT has to be calculated for. Calls of subroutines via variables or parameters have to be substituted by explicit subroutine calls.

- In third generation programming languages every semantic that can be programmed with gotos can also be achieved with the standard language constructs — sequences, loops, and iterations. As a consequence, the elimination of gotos does not result in any restrictions on programming.

- Since loops are fundamental for the implementation of almost every algorithm, one cannot eliminate loop constructs from programming languages. Nevertheless it has to be guaranteed that every loop terminates within a specified amount of time. As a consequence loop constructs which force the programmer to give some information about the time spent in a loop have to be introduced. Loops of this kind are called *bounded loops.*

  We differentiate two kinds of bounded loops:

1. loops with a specified limit for the number of iterations and
2. loops which are bounded by a time limit that must not be overrun at run-time.

The bound of each loop depends on its particular application context. It is specified using the appropriate language construct. Both limits for the maximal number of iterations and time limits have to be known at compile time to make the computation of the maximum execution time possible.

2.1 Bounded Loops

The constructs for bounded loops look very similar to conventional loop constructs. They differ from the usual loop constructs in two ways:

- All loop constructs enforce that a loop bound is specified — this has first been demanded in [Ehre 83] and [Hala 84]. A loop bound can either be a limit for the maximum number of iterations or a time limit for the termination of the loop. Loop bounds have to be known at compile time.
- If a loop bound is overrun a specified action is started. The default for this action is the activation of the operating system's exception handler (see section 2.2). However the programmer may override this default and specify a different treatment for this case[1].

The benefit of using bounded loops is twofold. On the one hand they are necessary for MAXT calculation, on the other hand they serve as a control mechanism for checking iteration limitations at run-time.

Examples for bounded loop constructs are demonstrated below. The constructs are presented in the C like syntax as used in a prototype implementation of MARS. In contrast to the original C loop constructs the keywords of bounded loops are written in capital letters. We provide FOR, WHILE and DO-WHILE loops which are derivates of the respective C loops.

FOR(*expr1*;*expr2*;*expr3*) MAX_COUNT(*const_expr*)

[1]It is a conceptual decision whether the action provided for an overrun of a loop is treated as an exception handler or as a feature for expressing a special timing behaviour of a program. The timing behaviour of the overrun action is only of interest in the latter case — when the action is part of the regular program (this is our interpretation). In case of an exception the timing behaviour cannot be guaranteed (see Exceptions).

stmt1

[ON_OVERRUN *stmt2*]

FOR(*expr1*;*expr2*;*expr3*) MAX_TIME(*const_texpr*)

stmt1

[ON_TIMEOUT *stmt2*]

The statements of the loop body are executed as long as the evaluation of the running condition (*expr2*) returns true and the loop bound — defined with MAX_COUNT or MAX_TIME — is not exceeded. If the running condition returns false the loop terminates and the program execution continues after the loop statement. If the running condition is true and the loop bound is found violated then either the statement in the ON_OVERRUN or ON_TIMEOUT clause or — if such statements do not exist — an operating system exception will be executed.

In our implementation the bounds of loops with an iteration limit are checked by a counter. In order to guarantee that the limits of time bounded loops are met we determine the maximal amount of time it takes to perform exactly one iteration. Every time a new iteration is to be started it is tested whether another iteration of maximal duration can be finished within the time limit or not.

We prefer this approach to the solution described in [Klig 86]. In [Klig 86] the time limit is transformed into an iteration bound at compile time. If the time needed for the single iterations is short compared to the maximal duration, the loop executes only a fraction of the specified time. We consider this a disadvantage, because in many cases the actual timing behaviour of loops will substantially differ from the behaviour specified in the program.

2.2 Exceptions

An exception is an abnormal situation during the execution of a program. We distinguish two kinds of exceptions — recoverable and non-recoverable exceptions.

Exceptions which are recoverable, i.e. an exception handler is provided, are considered part of the program. Their timing behaviour has to be taken into account in the timing analysis the same way as the timing behaviour of all program parts [Hala 89].

Non-recoverable we call exceptions which the system is not prepared for, i.e. if a

non-recoverable exception is raised the specified behaviour of the system cannot be guaranteed. Thus the occurrence of a non-recoverable exception leads to a system failure. All that can be done in this case is to try to minimize the resulting damage. The system has to be transferred into a save state and shut down afterwards.

3 Calculating an Upper Bound for the Maximum Execution Time

Programs which do not violate the conditions of section 2 and contain only simple constructs can be analyzed by an automated MAXT analysis tool. The maximum execution time can be calculated recursivlely using a small set of formulae for the simple language constructs. The simple language constructs for which formulae have to be provided are simple statements, statement sequences, alternatives, bounded loops, and subroutines.

The maximum execution time of a simple language construct (e.g. simple construct or simple expression) is the time required for the sequential execution of the corresponding machine instructions on the given processor. It can be obtained from the hardware specifications of the processor.

In order to calculate the $\mathrm{MAXT_C}$ for sequences ($construct_1$; $construct_2$; ...; $construct_n$;) we only have to sum up the maximum execution times of the single constructs. If we have to determine the maximal amout of time consumed by an alternative (`if` *condition* `then` $construct_1$; `else` $construct_2$;), we have to add the maximal time for the evaluation of the condition and the time maximally spent in one of the branches. The same rule can be used to calculate the maximum execution time for multiple branch instructions (switch in C, case in Pascal).

When calculating the maximum execution time for loop constructs, we have to distinguish between loops with a limited iteration number and loops which are bounded by a time limit. Dealing with the first kind the number of iterations has to be multiplied with the execution times of loop condition and body[2]. For loops with the running condition at the heading (FOR, WHILE) we have to add the time for one more evaluation of the condition. In the case of a FOR loop the time for initializations also has to be taken into account.

The maximum execution time of a time bounded loop is simply the timing constraint of the loop.

[2]Note that for bounded loops the overhead for testing the bounds also has to be regarded in the calculation of the maximum execution time

construct	*MAXT*		
primitive	$maxt(primitive)$	$=$	$\tau(primitive)$
sequence	$maxt(sequence)$	$=$	$\sum_i maxt(construct_i)$
alternative	$maxt(alternative)$	$=$	$maxt(condition)+$ $\max(maxt(construct_1), maxt(construct_2))$
$\text{loop}_{\text{number}}$	$maxt(loop_{head})$	$=$	$maxt(init) + maxt(condition)+$ $count * (maxt(body) + maxt(condition))+$ $maxt(overrun\_statement)$
	$maxt(loop_{tail})$	$=$	$count * (maxt(body) + maxt(condition))+$ $maxt(overrun\_statement)$
$\text{loop}_{\text{time}}$	$maxt(loop_{time})$	$=$	$time + maxt(timeout\_statement)$
subroutine	$maxt(subroutine)$	$=$	$\tau(organization) + maxt(body)$

Table 1: Formulas for the calculation of the maximum execution time

For both kinds of loops we have to consider the case that the loop bound is exceeded, which means that the loop has consumed its maximum time and the overrun or timeout statement is activated. Hence we have to add the maximum execution time for this statement.

The maximal time used up by a subroutine is the sum of the time for organization (copying parameters, jump, return from subroutine) plus the maximal amount of time for executing the subroutine body.

Table 1 summarizes the rules for the MAXT calculation for all constructs.

The function τ takes a simple action — simple statement, simple expression or subroutine organization whose MAXT is directly derived from the number of machine instruction cycles — as an argument and returns the amount of time it takes to execute this action in accordance to the prerequisites of definition 1. The *maxt* function is defined to calculate an upper bound for the execution time of its argument.

4 Adding Knowledge to Programs in order to Improve the MAXT Computation

Using the rules introduced so far we are able to compute an upper bound for the $MAXT_C$ of programs. We will demonstrate this in the following example. The example shows that our results can be improved if we add some more information about constraints on the control flow to the program. This will encourage us to introduce new language constructs.

4.1 An Example

An enterprise specializes in the production of goods that are filled into tin cans. The fabrication is automated and robots are involved in the job. In the last production step the cans are packed into boxes. We observe the following scenario: The tin cans are transported on a conveyor belt. A robot arm seizes the cans and puts them into the boxes. The computer controlling the robot arm is connected to a video camera in order to determine the position of the can on the belt.

The program for the localization of the can has to regard the specifications listed below:

- The image read by the camera is presented in an array of $640 * 200$ pixels.
- The colors of the tin cans and the conveyor belt are contrasting. Due to shape and size, cans produce an image that covers a maximum of 2200 pixels.
- There may be some noise in the image data. The maximal noise ratio which is tolerated is specified with one per cent (i.e. 1280 pixels) of the array.
- In order to steer the robot arm to the right location the program has to calculate the center of the marked area. The influence of noisy data on the result has to be minimized.

The program which performs as described is given by the Nassi-Schneidermann diagram shown in figure 1.

A listing for the camera application is shown on page 12 (the keyword `SCOPE` and the statement `MAX_COUNT(MAX_AREA);` should be ignored at this time. They will be described later in this paper). The execution times for all parts of the program calculated from the machine instructions generated by a UNIX C compiler are provided in comments[3]. They are given in number of CPU cycles.

[3]The program has been compiled with the portable C compiler on a 68000 machine

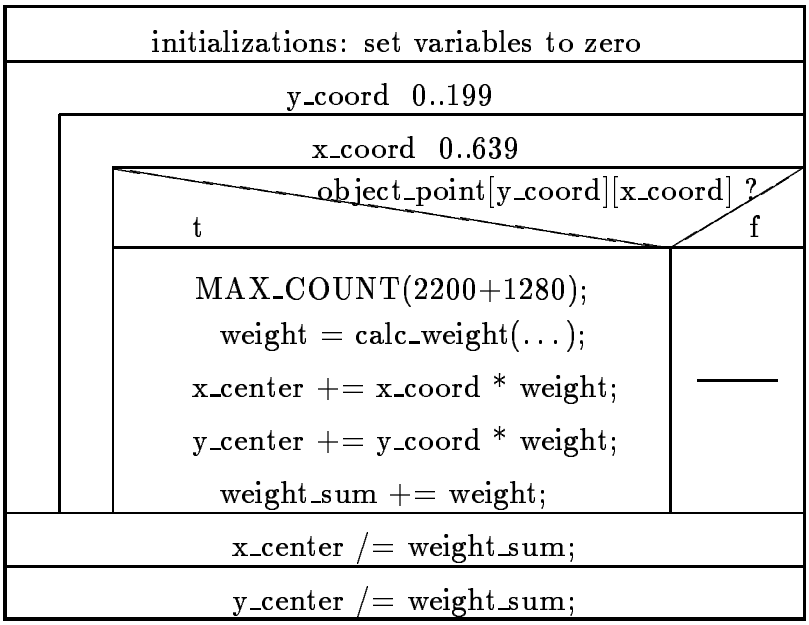

Figure 1: Nassi-Schneidermann diagram for the camera example

We can calculate an upper bound for the maximum execution time for the *calc_center*-subroutine by applying the introduced rules (see figure 3).

Looking at the program specifications we realize that the number of pixels that may be set is limited to 3 480(= 2 200 + 1 280). This knowledge is not contained in the program because currently we have no means to state that the respective program part can be executed a maximum of 3 480 times. Hence, the above calculation does not make use of this knowledge but assumes that each of the 128 000 pixels might be set, which results in a $MAXT_C$ high above a realistic execution time for this subroutine.

4.2 New Constructs: Markers and Scopes

The above example demonstrates that we can only determine an extremely high upper bound for the $MAXT_A$, if we are content with the language constructs introduced so far. The unsatisfactory results are due to the fact that we cannot express our full knowledge about the control flow of a program by means of the existing constructs. Thus we define new language constructs — markers and scopes — in order to reduce the difference $MAXT_C - MAXT_A$.

Definition 3 (Scope) *A Scope is a part of a program's instruction code, limited by a special scope language construct, that is embedded into the syntax of a*

```
#define MAX_ROWS    200                                                  /* CPU cyles */
#define MAX_COLS    640
#define MAX_AREA    3480


int calc_weight(image, x_coord, y_coord)                                      /*  44 */
char    image[MAX_ROWS][MAX_COLS];
int     x_coord, y_coord;
{
    int x_lim, y_lim, i, j, count=0;                                          /*  72 */

    x_lim = x_coord + 1;
    y_lim = y_coord + 1;

    FOR(i=y_coord-1;i<=y_lim;i++) MAX_COUNT(3)         /* loop1 */          /* (54+26;84;32) */
        FOR(j=x_coord-1;j<=x_lim;j++) MAX_COUNT(3)     /* loop2 */          /* (54+26;84;32) */
            if (image[i][j]) count++;                  /* alt1  */               /* (146) 16 */

    count--;    /* Number of neighbours */                                    /* 122 */
    return count * count >> 2;
}


int calc_center(image, x_center, y_center)                                    /*  44 */
char    image[MAX_ROWS][MAX_COLS];
int     *x_center, *y_center;
{
    int pixel_count, x_coord, y_coord, x_sum, y_sum;                          /*  48 */

    pixel_count = x_sum = y_sum = 0;

    FOR(y_coord = 0; y_coord < MAX_ROWS; y_coord++) SCOPE MAX_COUNT(MAX_ROWS)   /* (42+26;76;32) */
    {                                              /* loop3 */
        FOR(x_coord = 0; x_coord < MAX_COLS; x_coord++) MAX_COUNT(MAX_COLS)     /* (42+26;76;32) */
        {                                          /* loop4 */
            if (image[x_coord][y_coord])           /* alt2  */                /* 146 */
            {
                int weight;

                MAX_COUNT(MAX_AREA);                            /* marker */     /* 40 */
                weight = calc_weight(image, x_coord, y_coord);                /* 310 + */
                x_sum += x_coord * weight;
                y_sum += y_coord * weight;
                pixel_count += weight;
            }
        }
    }

    if (pixel_count)                               /* alt3  */                /*  22 */
    {
        *x_center = x_sum / pixel_count;                                      /* 424 */
        *y_center = y_sum / pixel_count;
    }
    else
        *x_center = *y_center = 0;                                            /*  56 */

    return 1;                                                                 /*  14 */
}
```

Figure 2: Listing of the camera example program

$$
\begin{array}{lcl}
maxt(calc\_center) & = & 44 + 48 + maxt(loop_3) + maxt(alt_3) + 14 = \\
 & = & \underline{551\ 475\ 096} \\
maxt(loop_3) & = & 42 + 76 + 200 * ((maxt(loop_4) + 32) + 76) + 26 = \\
 & = & 551\ 474\ 544 \\
maxt(loop_4) & = & 42 + 76 + 640 * ((maxt(alt_2) + 32) + 76) + 26 = \\
 & = & 2\ 757\ 264 \\
maxt(alt_2) & = & 146 + \max(310 + maxt(calc\_weight), 0) = 4\ 200 \\
maxt(alt_3) & = & 22 + \max(424, 56) = 446 \\
 & & \\
maxt(calc\_weight) & = & 44 + 72 + maxt(loop_1) + 122 = 3\ 744 \\
maxt(loop_1) & = & 54 + 84 + 3 * ((maxt(loop_2) + 32) + 84) + 26 = 3\ 506 \\
maxt(loop_2) & = & 54 + 84 + 3 * ((maxt(alt_1) + 32) + 84) + 26 = 998 \\
maxt(alt_1) & = & 146 + \max(16, 0) = 162
\end{array}
$$

Figure 3: MAXT calculation for the camera example

programming language.

Definition 4 (Marker) *A Marker is a special mark located within a scope. It specifies the maximal number the marked position in the program may be passed by the program flow between entering and leaving the scope.*

Using markers we can state the maximal number of times the control flow can pass through a specified position within a special part of a program designated by the scope construct. An arbitrary number of markers may be set in each scope.

Markers are mainly used to state that the number of executions of one or more paths through a loop can be bounded. It does not make sense to locate a marker in a sequence or a branch of an alternative which does not lie in a loop, because these program parts could be passed at most once within the scope. Hence, we design our scope language constructs to coincide with loop constructs.

We have to make a restriction for the use of markers inside scopes in order to avoid a complexity explosion as it might happen if all language constructs could be used within a scope arbitrarily. In loops, in which the maximal number of iterations is limited, markers (1) must not lie inside a piece of code which is contained by a loop (2) that is part of an alternative (3) (see figure 4). They can only be used in statement sequences or randomly nested alternatives which are only embedded in arbitrarily nested loops (limited by an iteration-bound) inside the scope (4).

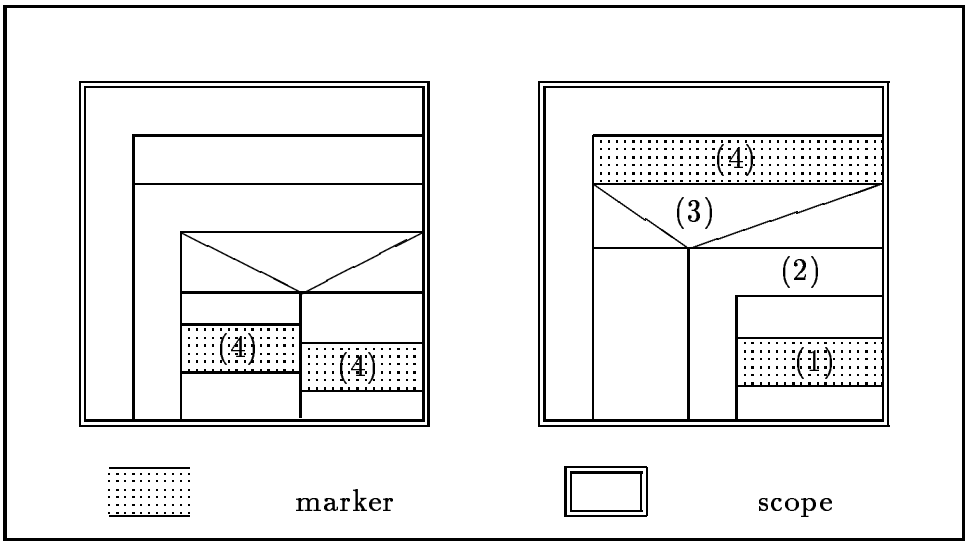

Figure 4: Example for validly (4) and invalidly (1) set markers (each of the diagrams represents a scope).

In time bounded loops markers may only be used at the highest level. This means that it is not allowed to set a marker inside nested loops which are bound by a time limit. An implementation of this feature would necessitate complex tests and thus evoke a substantial overhead at run-time.

Scopes are embedded in MARS-C, our programming language, which is a descendant of the C programming language, as an extension of the syntax of bounded loop constructs. The programmer defines a scope containing a bounded loop writing the SCOPE keyword into the loop's head. The extended syntax of FOR loops is shown below (the extensions for WHILE and DO-WHILE loop are equivalent). As the syntax suggests scopes can be nested arbitrarily. In the case of nested scopes a marker always refers to the innermost scope it is contained in.

```
FOR( expr1; expr2; expr3 ) [SCOPE] MAX_COUNT( const_expr)
        stmt1
[ON_OVERRUN stmt2]

FOR( expr1; expr2; expr3 ) [SCOPE] MAX_TIME( const_texpr)
        stmt1
```

[ON_TIMEOUT *stmt2*]

If a marker is to be set, there must always be an explicitly defined scope that it is contained in. We prefer this approach to the solution of a global, implicitly defined scope which contains all markers that are not inside any explicitly defined scope, since it leads to a more transparent behaviour and helps to detect errors. Furthermore, we favour the homogenous concept that every scope has to be defined explicitly with its contained loop.

To set a marker we simply have to write

MAX_COUNT(*const_expr*);

The constant expression *const_expr* can be evaluated at compile time. It is the maximal number of times the marker may be passed inside its scope. Markers are checked at run-time in order to verify that the calculated timing behaviour can be met. If the bound of a marker is violated an exception is raised.

Let us have another look at our example. We can improve the result of the code analysis by the use of markers and scopes. Now we need the code fragments which have been ignored when first looking at the listing on page 12. We define a scope which contains the loop in the *calc_center*-subroutine with the keyword SCOPE. Furthermore we set a marker MAX_COUNT(MAX_AREA);. The MAXT calculation for the new program is sketched in figure 5, section 6 (for MAXT calculation rules for markers and scopes see section 5).

4.3 Loop Sequences

Many programs use sequences of loops whose maximal iteration numbers complement each other, i.e. the sum of the iterations of the loops does not exceed a constant value at execution time. This kind of behaviour becomes interesting if the bound for the iteration sum is smaller than the sum of the maximal iteration numbers of the single loops. We can reach a better result for the MAXT calculation in such cases, if we are able to inform the MAXT analysis tool about loops that belong together.

Definition 5 (Loop Sequence) *A Loop Sequence is a series of loops (limited by an iteration limit) which have the property that the sum of the iterations of the single loops does not exceed a given constant value at run-time.*

Loop sequences are marked by an additional construct which is embedded into the extended C syntax.

```
LOOP_SEQUENCE ITERATION_SUM(const_expr )
        stmt1
[ON_OVERRUN stmt2]
```

The body of the loop sequence (*stmt1*) contains the loops that are part of the sequence and arbitrary other constructs — even loops — that may be programmed between the single loops. Loops which are members of a loop sequence are identified by the keyword `IN_SEQUENCE`. These loops must not be scoped since this would raise the complexity of programs immensely both for programmers and the MAXT analysis tool as well. The syntax definition for bounded loops, occuring in loop sequences, is a modification of the known constructs (see below).

```
FOR(expr1;expr2;expr3 ){IN_SEQUENCE | SCOPE}
MAX_COUNT(const_expr)
        stmt1
```

A bounded loop inside a loop sequence must not contain an `ON_OVERRUN` clause, since a violation of a loop bound inside the sequence is treated as an exception of the whole loop sequence. Loop sequences may provide an `ON_OVERRUN` statement.

5 Calculating the MAXT of Programs Using the Language Constructs Introduced

The gain of markers, scopes, and loop sequences is obvious by intuition. Here we want to give definite formulae for the calculation of the maximum execution time of these constructs.

5.1 MAXT Calculation for Scopes Containing Markers

In the following we provide some formulas for the calculation of the $\mathrm{MAXT}_{\mathrm{C}}$ of a scope containing n nested loops $loop_1, \ldots, loop_n$ with the innermost loop contain-

ing an arbitrary set of markers[4]. Each loop $loop_i$ contains all $loop_j$ with $i < j \leq n$. The maximal number of iterations of the loops are specified by $bound_1, \ldots, bound_n$. $loop'_j$ specifies the whole loop $loop_j$ except for the loops $loop_{j+1}, \ldots, loop_n$ in it.

The maximum execution time calculation for scopes with markers can be performed in two steps:

1. In the first step we calculate the maximum execution time of the loop parts ($loop'_j$) that do not contain markers, i.e. the time for the evaluation of loop conditions, loop organization and statement sequences between nested loops[5].

2. In the other step we treat the constructs of the innermost loop's body which may contain paths that are restricted by markers.

For both steps the maximum number the innermost loop body can actually be entered — $gmax$ — is needed. This figure can be calculated out of the bounds ($count_i$) of the n single loops and the restriction that markers place on the number of iterations ($mmax$).

$$gmax = \min(\prod_{i=1}^{n} count_i,\ mmax)$$

1. The loops have to be passed in a way that makes the execution time of the loops maximal. Hence, in our calculation we have to assume that the outer loops are passed as often as possible. All inner loops are executed in each iteration of a loop surrounding them.

   Deriving the contribution of the j-th loop to the MAXT of a scope we have to distinguish between three situations[6]:

   - $\prod_{i=1}^{j} count_i \leq gmax$

     This means that the loop is an "outer loop". In the worst case it will be entered $\prod_{i=1}^{j-1} count_i$ times, every time performing $count_j$ iterations. The maximal time consumption for this loop is

     $$\prod_{i=1}^{j-1} count_i \ * \ maxt(loop'_j(count_j))$$

[4]Note that this is a special case. More generally the single loops may contain an arbitrary number of loops with markers, and markers may be set at any level. The calculation model for this generalization however is only a slight variation of the model provided.

[5]For simplicity the time consumed by `OVERRUN`-statements has been neglected

[6]If $mmax \geq gmax$ all loops belong to the first group

- $\prod_{i=1}^{j-1} count_i < gmax \ \wedge \ \prod_{i=1}^{j} count_i > gmax$

  If this condition holds for a loop, the loop will both iterate more than once and less than a maximal number of times at least once in the worst case. The MAXT of this loop — obviously only one such loop can occur in each scope — can be calculated with the formula

  $$\sum_{1 \leq k \leq \prod_{i=1}^{j-1} count_i ; \sum_k count_{j_k} = gmax ; count_{j_k} \leq count_j} maxt(loop'_j(count_{j_k}))$$

  We consider it essential to mention that the maximal time for the overhead of $loop_j$ cannot be simply calculated as $maxt(loop'_j(gmax))$. This is due to the fact that it makes a difference whether a loop is entered $\prod_{i=1}^{j-1} count_i$ times or only once. A FOR-loop, for example, has to be initialized every time it is entered.

  Also note that we do not have to care about the number of iterations of the single instances of the loop — $count_{j_k}$. For a constant number of loop instances the $\mathrm{MAXT_C}$ is all the same if the sum total of the iterations ($\sum_k count_{j_k}$) is constant ($gmax$). It should be mentioned that the condition $count_{j_k} \leq count_j$ can be omitted. It has been introduced in order to establish a parallel to the loop's execution model.

- $\prod_{i=1}^{j-1} count_i \geq gmax$

  In this case the surrounding loops reach the maximal possible number of iterations. This means that in the worst case $loop_j$ can iterate maximally once each time it is executed. Consequently the contribution of $loop'_j$ to the MAXT is

  $$gmax \ * \ maxt(loop'_j(1))$$

2. In order to calculate the MAXT of a loop body with markers we build a graph which reflects its timing behaviour. Constructing the graph, program constructs that do not contain a marker are reduced — the MAXT for these constructs is calculated applying the known rules.

   The resulting graph consists of nodes representing branches or joins of program paths and edges which are marked with the MAXT of the appropriate program parts and execution restrictions imposed by markers.

The MAXT for the graph is calculated in a repeated execution of the two steps outlined below:

- The graph is searched for the longest path that has not been marked by the algorithm yet.
- The path which has been found in the first step is marked and the MAXT of the path is added to the MAXT of the whole graph.

5.2 Calculating the MAXT for Loop Sequences

In order to compute the maximum execution time for loop sequences we have to distribute the global maximal iteration number among the single loops, maximizing the sum of the MAXTs of the loops. We can apply the formula

$$maxt(loop\_sequence) = \max(\sum_{count_j \leq bound_j,\ \sum_j count_j\ \leq bound_{seq}} maxt(loop_j(count_j)))$$

where $bound_{seq}$ is the global maximal number of iterations in the loop sequence, $bound_j$ are the upper iteration bounds of the single loops and $count_j$ the actual bounds. $loop_j(count_j)$ stands for the time it maximally takes to execute $count_j$ iterations of loop $loop_j$.

6 Benefits and Costs of the New Constructs

On page 15 we introduced some changes to the example of section 4.1. We put a scope around the main loop and introduced two markers in the program listing. Based on the rules for the MAXT calculation of these constructs that have been described, we want to give a demonstration of their benefits. Figure 5 shows the steps in the calculation of the maximum execution time for the modified program fragment.

The main gain is due to the marker. With the marker we express that the time consuming procedure *calc_weight(...)* is not called in every iteration (128 000 times) but at most 3 480 times (this is the maximal number of pixels which may be set according to the specifications). Thus the portion of the MAXT_C which is contributed by the statement sequence containing the procedure call is immensely reduced in comparison to the first version of *calc_center(...)*.

The upper bound value calculated for the maximum execution time of the original example was about 551 million CPU cycles. When analyzing the new

$$
\begin{aligned}
maxt(calc\_center) &= 44 + 48 + maxt(scope) + maxt(alt_3) + 14 = \\
&= \underline{46\ 810\ 232} \\
maxt(scope) &= maxt(loop'_3) + maxt(body') = 46\ 809\ 680 \\
maxt(loop'_3) &= 16 + 42 + 76 + 200 * ((maxt(loop'_4 + 32) + \\
&+ 76) + 26 = 13\ 874\ 560 \\
maxt(loop'_4) &= 42 + 76 + 640 * ((0 + 32) + 76) + 26 = 69\ 264 \\
maxt(body') &= 200 * 640 * 146 + 3480 * (40 + 310 + maxt(calc\_weight)) = \\
&= 32\ 935\ 120 \\
maxt(calc\_weight) &= 3\ 744 \\
maxt(alt_3) &= 22 + \max(424, 56) = 446
\end{aligned}
$$

Figure 5: MAXT calculation for the enhanced camera example

version we derive a bound of only 47 million cycles for the program execution. So the calculated MAXT is reduced by a factor of more than $\underline{10}$ by the use of a scope and a marker. Although this result is remarkable we cannot generalize it. This is due to the fact that the reduction depends on the complexity of the program parts involved (the complexity of *calc_weight* has a significant impact on the improvement factor in our example).

It is also interesting to remember that markers are a means not only for an improvement of the calculation of the MAXT but also for the supervision of the correct program behaviour. As the example shows plausibility checks which are based on application specific knowledge can be built into programs. In this way failures in the program logic may be detected at run time so that the self checking property of programs can be improved.

Every marker is a kind of cheap investment that pays for itself. Checking a marker costs 40 CPU cycles every time the marker is passed. These 40 cycles have to be invested only as often as the marker is passed, not for the other iterations of the loop. The time for a marker's evaluation will be outweighed by the benefits its information brings for MAXT calculation. The gain will be the greater the more a marker restricts the number of iterations through a program part and the longer this program part takes to execute.

When calculating the time overhead for the markers of our example we got a number of less than 140 000 cycles which is only about 2% of the total overhead in the example. The gain of introducing these constructs can be extracted from figures 3 and 5 and is summarized in table 2 — 504 million CPU cycles could be saved and a processor idle time of 91% could be prevented by reducing the

$MAXT_C$ of the two versions of the example:

bounded loops only	551 475 096
bounded loops and markers	46 810 232

Overhead for bounded loops and markers:

bounded loops (calc_weight)	2 115 840	
bounded loops (calc_center)	4 879 238	
marker	139 216	
bounded loops and markers	7 134 294	cycles

Table 2: CPU cycles and overhead for the camera example

calculated maximum execution time to a realistic value. So the investment has proved worth while.

7 A Concept for a MAXT Analysis Tool

In the previous sections we introduced some new language constructs for real-time programming languages. The costructs were defined in a C like syntax as we use them in MARS-C, which is an extension of the C programming language [Kern 78]. Besides the constructs mentioned in this paper MARS-C also provides statements for sending and receiving messages in the distributed real-time system MARS [Pflü 89].

The calculation of the maximum execution time of a MARS-C program is done in two steps, a compilation and an analysis step. In the first phase the program is compiled with the MARS-C precompiler. The result of the precompilation is a program in C source code with some additional information about loop bounds, markers, and loop sequences. The C program then is compiled into assembler code.

In the analysis step (the tool for the calculation of the MAXT is in developement) the real calculation of the upper bound for the applicaton specific maximum execution time of the program takes place. This calculation also consists of two parts.

- In a first run information about program structure and timing behaviour has to be combined. In this step the result of the precompiled MARS-C source code and the assembler source are parsed. As an intermediate result a file is created that includes full information about the program structure (alternatives, loops, loop limits, markers, scopes, and loop sequences) and the execution time for all parts, derived from the execution times of the single assembler instructions.

- In the second step of the analysis the intermediate file is read and the MAXT of the program is calculated. The calculation is based on the formulae described in this paper.

Representing the intermediate results of the analysis step in a file has two advantages: In the developement phase of the tool the file can easily be read so that results can be verified. When the tool is in use the file gives the programmer detailed information about the timing properties of all parts of his program.

8 Summary

In this paper we discussed the aspects of a source code based calculation of the maximum execution time of tasks. We introduced restrictions for analyzable programs and presented formulae for the MAXT calculation for the fundamental language constructs of third generation programming languages.

Comparing the calculated maximum execution time of tasks to the actual maximum execution time we observed large differences in some cases. The calculated upper bound for the maximum execution time of programs was much higher than the actual maximum execution time. We managed to reduce this gap by the introduction of new programming language constructs — markers, scopes and loop sequences — that allow programmers to utilize knowledge about the execution of their algorithms.

It has also been pointed out that markers can be used as a means of flow monitoring. Application specific knowledge can be expressed in the program code directly so that the self checking property of programs can be improved.

A simple example demonstrated the gains of the new programming means. The MAXT value for the example program was decreased by a factor of 11. Considering that the example is relativly simple we expect that the new constructs can yield even greater gains when the complexity of the application increases.

References

[Darl 78] J. Darlington, R. M. Burstall
A System which Automatically Improves Programs
in
David Gries
Programming Methodology
Springer, New York, 1978

[Ehre 83] W. Ehrenberger
Softwarezuverlässigkeit und Programmiersprache
Regelungstechnische Praxis, 25. Jhrg., Nr. 1, 1983, pp. 24–29

[Fohl 88] G. Fohler and Ch. Koza
Scheduling eines verteilten Echtzeitsystems mittels heuristischer Suchstrategien
Research Report No. 1/88, Institut für Technische Informatik, Technical University of Vienna, Jan. 1988

[Hala 84] W. A. Halang
A Proposal for Extensions of PEARL to Facilitate the Formulation of Hard Real-Time Applications
Proceedings "Fachtagung Prozeßrechner 1984", Karlsruhe, Sept. 1984
Informatik-Fachberichte 86, Springer, Berlin-Heidelberg-New York-Tokyo, 1984, pp. 573–582

[Hala 89] W. A. Halang
A Priori Execution Time Analysis for Parallel Processes
Proceedings of the Euromicro Workshop on Real-Time, Como, June 1989, Washington, IEEE Computer Society Press, 1989

[Kern 78] B. W. Kernighan, D. M. Ritchie
The C Programming Language
Prentice Hall, New Jersey, 1986

[Klig 86] E. Kligerman, A. D. Stoyenko
Real-Time Euclid: A Language for Reliable Real-Time Systems
IEEE Transactions on Software Engineering, Vol. SE-12, Number 9, Sept. 1986, pp. 941–949

[Kope 89] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, R. Zainlinger

Distributed Fault-Tolerant Real-Time Systems: The MARS Approach
IEEE Micro, Feb. 1989

[Lein 80] D. W. Leinbaugh
Guaranteed Response Times in a Hard-Real-Time Environment
IEEE Transactions on Software Engineering, Vol. SE-6, Number 1, Jan. 1980, pp. 85–91

[Lein 82] D. W. Leinbaugh, M.-R. Yamini
Guaranteed Response Times in a Distributed Hard-Real-Time Environment
Proceedings of Real Time Systems Symposium, IEEE Press, Dec. 1982, pp. 157–169

[Lein 86] D. W. Leinbaugh, M.-R. Yamini
Guaranteed Response Times in a Distributed Hard-Real-Time Environment
IEEE Transactions on Software Engineering, Vol. SE-12, Number 12, Dec. 1986, pp. 1139–1144

[Liu 73] C. L. Liu and J. W. Layland
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment
Journal of the ACM, Vol. 20, Number 1, Jan. 1973, pp. 46–61

[Mok 84] A. K. Mok
The Design of Real-Time Programming Systems based on Process Models
Proceedings of Real Time Systems Symposium, IEEE Press, Dec. 1984, pp. 5–16

[Pflü 89] M. Pflügl, A. Damm, W. Schwabl
Interprocess Communication in MARS
Proc. of the ITG/GI Conference on Communication in Distributed Systems, Stuttgart, Feb. 1989

[Senf 89] Ch. Senft, R. Zainlinger
A Graphical Design Environment for Distributed Real-Time Systems
Proceedings of the 22nd IEEE Conference on System Science, Kailua-Kona, Jan. 1989, Washington, IEEE Computer Society Press, 1989, pp. 871–880

[Stoy 87] A. D. Stoyenko
A Real-Time Language with a Schedulability Analyzer
Technical Report CSRI-206, Computer Systems Research Institute, University of Toronto, Dec. 1987