Incorporating Worst Case Execution Time in a Commercial C-compiler

H. Börjesson

January 8, 1996

t91hab@docs.uu.se
Undergraduate thesis work
Department of Computer Systems
Uppsala University

Abstract: This report presents a timing tool that calculates worst case execution times for programs written in C. The tool is integrated with IAR-Systems C-compiler for the 8051 microcontroller. For each function in the source code the tool calculates both the worst case and the best case execution times. It is also possible to add extra timers into the code to calculate execution times for certain parts of a function. These timers allow measurements of the execution times of small parts of a function. This may be useful in identifying bottlenecks in the code. The report also discusses and presents solutions to several general execution time analysis problems and reviews some theory available today together with a few existing timing tools.

Keywords: hard real-time systems, worst case execution time, disassembler, basic blocks, flow graphs.

Preface

This report is part of my undergraduate thesis work to achieve my Master of Science in engineering physics. The work was performed at the Department of Computer Systems, at Uppsala University, under supervision of Hans Hansson. The project is performed within the ASTEC (Advanced Software TEChnology) center of excellence supported by NUTEK (Swedish National Board for Technical and Industrial Development) in cooperation with IAR-Systems, supplier of C-compilers for microcontrollers.

This report is written in LaTeX, with 11 points, article style. The figures are drawn with xfig generating encapsulated postscript files. Bibtex is used for the bibliography.

Acknowledgments

I would like to thank my supervisor Hans Hansson for employing me, and for helpful notes about the layout of the report. I would also like to thank the rest of the crew working at DoCS (Department of Computer Systems). Special thanks to Fredrik Skoglöv, Harmen van der Velde and Thiemo Voigt for valuable comments. Very special thanks to Anders Berg at IAR-Systems and the rest of the staff at IAR. Thanks to all the scientist mentioned in the bibliography without whom this report would not have been written. Finally I would like to thank my girlfriend Malin Karlsson, who have supported me all the way.

Hans Börjesson

Contents

Part I
The WCET Timing Tool for the 8051 Microcontroller

1 Introduction

WCET (Worst Case Execution Time) has been in focus for several scientists during the last decade. You may ask: "why is it so important?" One reason is that if you want to guarantee (and predict) that a certain deadline will be met in a real-time system, you must have the WCET in advance. Programs with deadlines that *must* be met under all circumstances are often called **hard real-time programs**.

One example when execution time analysis can be useful is when deciding a systems configuration. Knowing the WCETs for different configurations it will be possible to choose hardware components that will meet the required performance. This might make the design cheaper and more efficient, since you can analyze your system, and hence you do not always have to buy the most expensive (and fastest) components to be assured that the timing requirements will be satisfied. Another example when WCET analysis is required is in scheduling analysis. The scheduling analysis known today require upper bounds on the WCET to be known in advance.

1.1 About the Report

Chapter 2 discusses timing tools in general, and explains some important concepts related to execution time analysis. In chapter 3 the 8051 microcontroller is presented. The 8051 is the target processor (microcontroller) for the WCET tool presented in this report. Chapter 4 presents a short look at the compiler and also discusses the pragmas used in the timing tool and also why they are used. Chapter 5 explains the data structures used and how the parsing of the pragmas is done, and finally discusses how the WCET calculation is performed. In Chapter 6 some test results are presented. Part two of the report discusses existing timing tools and theories to solve different kinds of problems, e.g. the way cache memories can be handled etc. In chapter 7 an empirical timing tool is presented. Analytical timing tools are the topic of chapter 8. In chapter 9 some conclusions are drawn and possible future work is discussed.

2 Important Concepts for WCET Analysis

In this section, a couple of important concepts (welcome to the jungle!) relevant to execution time analysis will be explained.

Timing tools can be divided into two broad categories: empirical timing tools and analytic timing tools. Empirical timing tools tries to measure the execution time by running the program. Analytic timing tools on the other hand tries to calculate the WCET without having to run the program. More about analytic and empirical timing tools will be presented in chapter 7 and 8.

2.1 Basic Blocks

Almost all algorithms used to calculate the WCET in analytical timing tools identifies something called **basic blocks** in the analyzed program. A basic block is a code segment with maximum length and with single entry and single exit points [AUS86]. If one instruction in a basic block is executed, then all will be. As an illustration, consider the following if-statement written in C.

```
1  if (a>b)
2  {
3    a++;
4    b++;
5  }
6  else
7  {
8    a--;
9    b--;
10   a=a-b;
11 }
```

The if-statement consists of three basic blocks. The first block is the expression defining the condition in line 1. The second one is line 3-4, and the third one is line 8-10. When calculating the WCET for the above code segment you can use the following formula:

WCET(if statement)=time(block1)+max(time(block2),time(block3))

i.e. the time it takes to execute line 1 (the condition) plus the maximum of block2 and block3 execution times, since either the if or the else part will be executed.

2.2 Flow Graphs

A flow graph is a graph that shows how a program can be executed. Each node in the graph represents a basic block. From a basic block B1 there is an edge to a block B2 if B2 can immediately follow B1 in some execution sequence. B1 is then called the *predecessor* of B2, and B2 is the *successor* of B1. In figure 1, you can see the flow graph for the if-statement given in section 2.1.

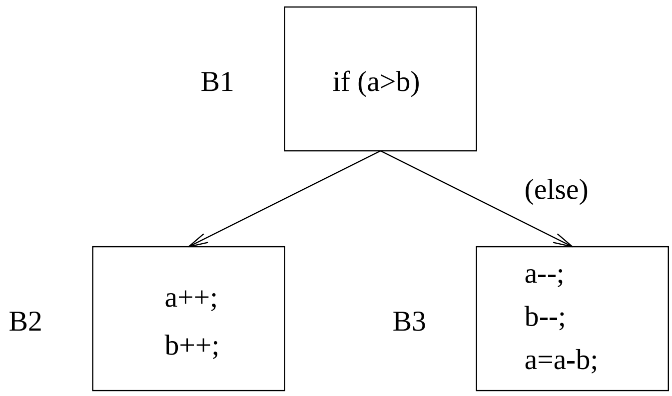

Figure 1: A simple flow graph of an if-statement

2.3 Restrictions and Difficulties

The computer systems used today were (are) not built with execution time analysis (or predictability in general) in mind. The most important issue is average case speed and efficiency. Some of the problems that have to be solved to obtain accurate WCET measures are presented below.

- The most obvious is loop-bounds. If you do not have a bound for the number of iterations of a *while*-loop (for example), you cannot say anything about the WCET, because the loop may be executed an infinite number of times.
- Recursion is also a problem, because you do not in general know the maximum depth of the recursion in advance. Recursion also leads to stack allocation problems. The maximal stack space needed at run-time cannot in general be predetermined.
- General use of *goto*-statements hinder analysis, because it may disrupt the program structure. With *goto*-statements it is possible to jump into (or jump out of) a loop-body without executing the "head" of the loop. This complicates execution time analysis.
- Cache memory is another problem. The reason for this is that using cache memory makes it difficult to make a good model for analysis. In general it is very difficult to know what is in the cache and what is not. A lot of research has been done in this area. Some of the theories available today will be presented in chapter 8.11.
- Pipelines are also difficult to handle because they are difficult to model accurately. Some work has been done in this area which will be presented in chapter 8.9.
- Dynamic memory allocation is not allowed by most timing tools, because dynamic memory tends to have unpredictable timing behavior.
- Various memory types with different access times is a problem since it is difficult to predetermine in which memory instructions and data are stored.
- The C *?-operator*[1] cannot be handled because in the compiler used *?-operators* do not have an ordinary flow graph (a feature of the compiler used). This make the analysis of these statements more difficult.

[1] ?-operator can sometimes be used instead of *if*-statement

Many of these problems may be solved by just giving the restriction not to use them. There are many processors (especially microcontrollers) not using pipelines, caches etc. *Goto* statements are in general banned by programmers. Loops on the other hand are frequently used program constructs, and it would be very difficult to write programs without them. One solution to this problem is to use min/max loopbound annotations, by which the programmer provides the min/max number of iterations of the loop. This makes it possible to calculate the execution time since you know the maximum number of iteration the loop is allowed to iterate and you also can get the execution time for one iteration. Using programmer provided loopbounds may solve some problems but they introduce the problem of making sure that they are true bounds. Note that also a *for*-loop may cause problems, as illustrated in the following example: (C-language).

```
for(i=0;i<temp;i++)
```

The variable "temp" is not known in advance. To achieve predictability loopbounds is needed also for *for*-loops of this kind.

3 The 8051 Microcontroller

The 8051 microcontroller was first developed by Intel in 1980. It is one of the most popular microcontroller used today. It is classified as an 8-bit machine since the internal ROM, RAM, special function registers, arithmetic/logic unit and the external data bus are each 8 bits wide [SIE88]. The 8051 is produced by several manufacturers and the controllers might have some differences, but a Siemens SAB 8051 contains [SIE88]:

- 8 Kbyte on-chip program memory
- 256 byte on-chip RAM
- Six 8-bit parallel ports
- Full duplex serial port, 4 modes of operation, fixed or variable baud rates
- Three 16-bit timer/counters
- A/D converter, 8 multiplexed analog inputs, programmable reference voltages
- A 16-bit watchdog timer
- Power-down supply for 40 byte of RAM
- Boolean processor, which can perform operations like jump-if-set, jump-if-not-set tec.
- 256 directly addressable bits
- 12 interrupt sources (7 external, 5 internal), 4 priority levels
- Stack depth up to 256 byte
- 1 μs instruction cycle at 12 MHz operation
- 4 μs multiply and divide
- External program and data memory expandable up to 64 Kbyte each
- Compatible with standard SAB 8080/8085 peripherals and memories
- Space-saving PLC-68 package

A “pin-out” symbol is showed in fig 2 where *port 0-8* are the six parallel ports. The *ALE* (Address Latch Enable) is used mainly to provide a properly timed signal to latch the low byte of an address from P0 to an external latch during fetches from external memory. *PSEN* (Program Store ENable) is used as a read strobe to access external program memory. *RESET* sets the internal registers to special predefined values. RAM is not affected by *RESET*. *XTAL1* and *XTAL2* are the input/output of an inverter. More details about the signals can be found in [SIE88].

The CPU can be split up in three pieces: the instruction decoder, the arithmetic section, and the program control section. The instruction decoder decode program instruction and generates internal signals controlling the functions of the individual units within the CPU. For example it controls the *ALU* (Arithmetic/Logic Unit) processing. The arithmetic section for the processor manipulate data and is comprised of an *ALU*, an *A* (Accumulator register) register, an *B* (used during multiply and divide) register and *PSW* (Program Status Word) register. The *ALU* can perform operations like subtract, multiply, divide, AND, OR etc. The *ALU* accepts 8-bit data words and generates an 8-bit result, There is also a boolean processor which can perform bit operations like set, clear, complement, jump-if-set, jump-if-not-set etc. The program control section controls the sequence the instructions are executed.

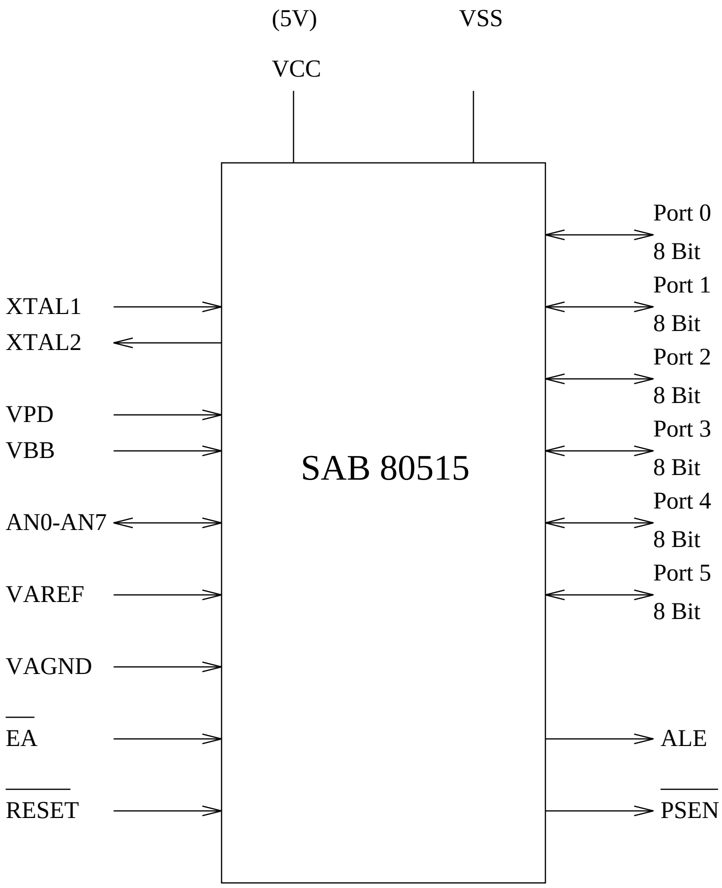

Figure 2: Logic pin-out symbol over the SAB 80515

3.1 The Instruction Set

The instruction set for SAB 80515 includes 111 instructions. 49 of them are single byte, 45 are two byte and 17 are three byte instructions. Each instructions consists of a function mnemonic followed by a destination and a source field. Some instructions are showed in table 1.

Mnemonic	Description	Byte	Cycle
Arithmetic operations			
ADD A,Rn	Add register to accumulator	1	1
SUBB A,@Ri	Subtract indirect RAM from A with borrow	1	1
INC Rn	Increment register	1	1
DEC A	Decrement accumulator	1	1
MUL AB	Multiply A and B	1	4
Logic operations			
ANL A,Rn	AND register to accumulator	1	1
XRL direct #data	Exclusive OR immediate data to direct byte	3	2
Data transfer			
MOV A,Rn	Move register to accumulator	1	1
MOV direct,A	Move accumulator to direct byte	2	1
PUSH direct	Push direct onto stack	2	2
POP direct	Pop direct from stack	2	2
Boolean variable manipulation			
CLR C	Clear carry flag	1	1
ORL C,bit	OR direct bit to carry flag	2	2
Program and machine control			
RET	Return from subroutine	1	2
AJMP addr11	Absolute jump	2	2
JNB bit,rel	Jump if direct bit is not set	3	2
NOP	No operation	1	1

Table 1: A subset of the instruction set for the SAB 80515

The simplicity of these chips, e.g. no pipelines, caches etc. makes them suitable for execution time analysis. This was one reason to choose the 8051 to be the target processor (microcontroller) for the tool presented in this report. Still, there are difficulties with the 8051. For example the differences in access time of its various memories, i.e. the execution time analyzer must know which access time to use.

4 A Short Look at the Compiler

The compiler used is IAR-Systems C-compiler for the 8051 microcontroller. The compiler can be divided into the following three modules, as illustrated in fig 3: the parser which parses

Figure 3: The C-compiler's three modules

the source code, the global optimizer which builds a flow graph with information of how basic blocks are connected to each-other and also performs some optimizations, and the code generator which generates the code and also performs some optimizations specific for the target processor, e.g. register optimization. The code generator also keep track of the execution times for the translated instructions for later use during the traversal of the program. The code generator also keeps track of where data and instructions are stored which eliminates the problem with different access times for different memories.

4.1 Why Use Pragmas?

To be able to measure the WCET in a program, some restrictions to the language have to be made, as explained in section 2.3. Another big problem with execution time analyzers is that their estimates tend to be pessimistic[2]. The main reason for this is that several of the analyzed execution paths will never be executed. We call these paths *infeasible*. An infeasible path is a path that theoretically can be executed but will never be executed at runtime. Most compilers are not capable of automatically recognizing these paths since it would require an elaborate semantic program analysis. Unfortunately, since infeasible paths are included in the WCET calculations unnecessarily high WCETs may be obtained. This problem can be tackled by annotations in the source code. Such annotations are often referred to as pragmas. As an illustration of the use of pragmas consider the following naive example:

```
        if (a>10)
        {
S1        code that take 100 clock-cycles to execute
        }
        else
        {
S2        code that take 25 clock-cycles to execute
        }

        if (a>5)
        {
S3        code that take 25 clock-cycles to execute
        }
        else
        {
S4        code that take 100 clock-cycles to execute
        }
```

The execution times for the different paths through the two *if*-statements are presented in table 2 (the time to check the conditions are excluded). The most time consuming path (call it path

	Statement S1	Statement S2
Statement S3	125 cycles	50 cycles
Statement S4	200 cycles	125 cycles

Table 2: Execution times for the two *if*-statements

1) through these two *if*-statement is to take the *if* part of the first *if*-statement (S1) and the *else* part of the second *if*-statement (S4) which will give an execution time of 100+100 clock-cycles (plus the cost of testing the conditions). If you do a semantic analysis of the two *if*-statements, it is easy to realize that path 1 never can be executed at runtime. The reason is that if the *if* part of the first *if*-statement is executed, **a** must be greater than 10. This means that **a** cannot execute the *else* part of the second *if*-statement (if **a** does not change its value between the

[2]here pessimistic means to overestimate the real WCET by a substantial margin

if-statements). If it were possible to incorporate extra information about the flow, infeasible paths would be removed which in general leads to tighter WCETs.

To help the parser in the compiler understand that a certain line in the source code is a timing pragma, a reserved word *T_info* is used. When the parser finds this word, it takes the rest of the line and sends it to a special pragma parser. This parser checks the pragma to see if it is valid[3], and if so, fills in the information given by the pragma in some internal data structures of the compiler. This information is later used during the traversal (the WCET calculation) of the flow graph. More details about these pragmas are given in chapter 4.2.1.

4.2 Pragmas

In this section we will describe the following pragmas introduced in the 8051 C-compiler to facilitate accurate execution time analysis. The pragmas can be divided into four categories: the label pragma, the loopbound pragma, the path pragmas and the timing pragmas. They are all described in the following subsections.

4.2.1 The Label Pragma

The *label* pragmas are names of certain statements (basic blocks). These names can be used by other pragmas to refer to these statements. If you for example have two *if*-statements with the two labels "if1" and "if2" and you know that these two statements never can be executed together, it is possible to declare a new pragma, which can use these labels and tell the compiler (and timing tool) that the paths including both "if1" and "if2" should not be included in the WCET calculation. An example of a *label*-pragma is presented below, which gives a label to an *if*-statement ("if1").

```
a++;
c++;
T_info label(if1)
if (a>c)
{
  c=a;
}
```

4.2.2 The Loopbound Pragma

To be able to take loops into account in the WCET calculation, they have to be limited in some way. An infinite loop is not allowed because it has an infinite execution time (if you allow infinite loops the WCET will also be infinite). One way of tackling this problem is to introduce loopbounds. For each loop (*while, for, do while* in C) a *loop_bound(min, max)* pragma is given, stating the minimum/maximum number of times the loop may iterate. This allows the execution time to be calculated since you compute the execution time for one iteration of the loop, and you also know the minimum/maximum number of iterations the loop is allowed to run. As an illustration consider the following example:

```
a++;
c++;
```

[3] Here valid means a correctly written and not multiple defined pragma

```
T_info loop_bound(5,15)
while (a<c)
{
  a++;
  c=c-2;
}
```

It shows a couple of C-statements including one *while*-loop and the associated *loop_bound* pragma. It is, of course, very difficult to set appropriate values for the loopbounds, but this is a price you have to pay to be able to use loop constructs. If the loopbound is broken at run time a warning will be given.

4.2.3 The Path Pragmas

As illustrated earlier in chapter 4.1, it is valuable to remove infeasible paths through conditional statements. This is done by using three different pragmas introduced in [PS91, Par93]. The pragmas are *same_path(label1, label2), not_same_path(label1, label2)* and *exclusive_path(label1, label2)*. The labels *(label1, label2)* are names of statements[4] as explained in chapter 4.2.1. The pragmas work as follows:

- *same_path(label1, label2)* is used if you know that the statement corresponding to *label2* (statement2) is always executed if the statement corresponding to *label1* (statement1) is executed. It is important to understand that statement2 may be executed even if statement1 is not executed (i.e there is a difference between *same_path(label1, label2)* and *same_path(label2, label1)*).
- *not_same_path(label1, label2)* is used when you know that statement1 (corresponding to *label1*) and statement2 (corresponding to *label2*) never can be executed together. It is possible that none of the statements are executed but never both of them.
- *exclusive_path(label1, label2)* is used when you know that either statement1 or statement2 is executed but never both or none.

These three path pragmas can be expressed as follows (A and B are two labels on two basic blocks).

- *same_path(A,B)*: $\neg(A \wedge \neg B)$
- *not_same_path(A,B)*: $\neg(A \wedge B)$
- *exclusive_path(A,B)*: $(A \wedge \neg B) \vee (\neg A \wedge B)$

Because the tool calculates WCET function wise the pragmas are only valid in that function. This means that you cannot calculate execution times from the middle of function *f(x)* to the middle of function *g(x)* since the program only calculates execution times per function. This can of course be solved by using different timers, as explained in chapter 4.2.4, and by adding the different functions execution times separately.

4.2.4 The Timing Pragmas

The WCET tool presented in this report will give the MCET (Min Case Execution Time) and the WCET for each function. If the programmer wishes to measure only certain parts

[4] A statement is part of a basic block so the label name is also the name of the basic block

of a function, it is possible to do this by using the *start(timer_name)*- and *stop(timer_name)*-pragmas. The *timer_name* is simply a unique name of the timer. The programmer inserts the *start*-pragma in the basic block where he wants the calculation to commence, and inserts the *stop*-pragma where he wants to stop the calculation. The execution times are not given for each statement, they are given for whole basic blocks. This means that it is not possible to calculate execution times within a basic block (but you can, of course, get the execution time for one basic block). The timers are valuable because they enable the programmer to test certain code pieces and thereby locate bottlenecks in the program.

5 Algorithm and data structures

This chapter presents the algorithm and data structures used. It is quite a technical chapter and it may be skipped by the reader the first time. First a short description how the execution times are being calculated, we then describe the data structures used to handle the different pragmas, and finally a more detailed algorithm description is given in chapter 5.4.

5.1 A Short Description of the Algorithm

To be able to calculate the WCET, all paths through the program must be passed. When a program is compiled the compiler constructs a flow graph (or several flow graphs). As mentioned earlier in chapter 2.2, the flow graph shows how basic blocks are connected. The flow graph is traversed and the execution times for all paths are calculated by using the execution times for each basic block given by the code generator. To remove infeasible paths the *path*-pragmas are used. When all paths are traversed, the paths with the WCET and the MCET are presented together with all execution times corresponding to timers (described in 4.2.4).

5.2 The Pragma Struct

All different kinds of pragmas are defined in a global structure called *array* which is a vector of *id_rec_T00* elements. The *id_rec_T00* struct is defined as follows:

```
typedef struct id_record_T00
{
  char                        *name;
  unsigned short              args;
  fp_T00                      *func;
  unsigned short              nr;
} id_rec_T00;
```

where

- *name* is a char pointer pointing to the name of the pragma.
- *args* is the number of arguments of the pragma.
- *Func* is a pointer to a unique parsing function for the pragma. The *fp_T00* is defined as follows: *typedef CVM_rec_D05 *fp_T00(char*, char*)* which means a function that take two char pointers as argument and returns a *CVM_rec_D05* pointer. *CVM_rec_D05* is an internal data structure representing a statement in the compiler.
- *nr* is the number of characters in the name.

This structure makes parsing pragmas quite easy and it is also easy to add more pragmas in the future by just adding an element to *array*.

5.3 The Pragma Parser

The compiler can be split up in three parts as described in chapter 4: the front end, the global optimizer and the code generator, (see fig. 4). The WCET Module is the module added to incorporate WCET analysis in the compiler. The front end parses the source code and each

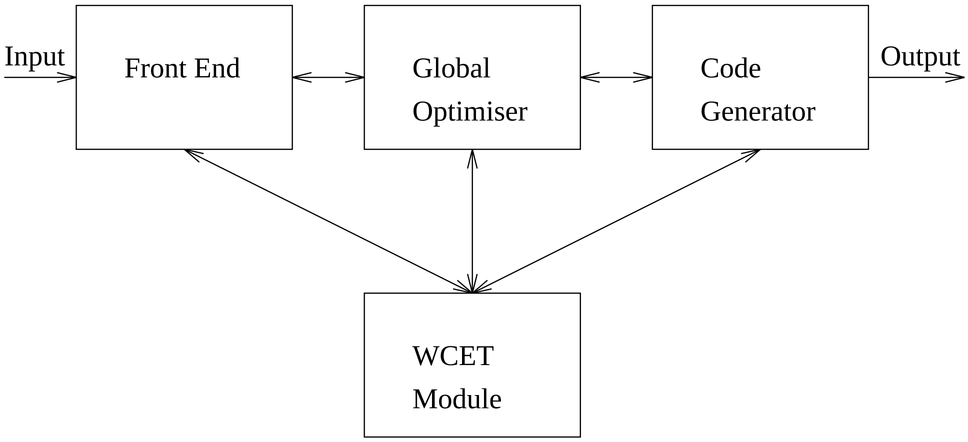

Figure 4: The C-compiler's three modules and the WCET module

time the parser finds a pragma, it is sent to a special parsing function in the WCET module. This parsing function checks if the pragma is valid[5]. If for example a *label*-pragma is already defined, an error will occur and the WCET analysis will stop. This pragma check is easy to perform by using the *array* vector described in section 5.2 and the *CVM_rec_D05* function defined below.

```
CVM_rec_D05 *parse(char *str)
{
  id_rec_T00 *ptr;
  char         *c;
  for (ptr = global_info; ptr->id; ptr++)
  {
    if (strncmp(ptr->id, str, ptr->nr) == 0)
    {
      c = (char*)get_memory_M30((strlen(str)-ptr->nr+2)*(sizeof(*c)));
      c = strchr(str,'(');
      c++;
      return ((ptr->func)(c,ptr->id));
    }
  }
  return 0;
}
```

The parse function takes a pragma as argument (pointed to by a char pointer) found by the front end of the compiler. This pragma is compared with all pragma definitions (stored in *array*) and if the found pragma matches one of the pragma definitions (checked in the *if*-statement) a pointer to the argument(s) of the pragma is created (c) and the corresponding parsing function is called.

In the global optimizer a flow graph is constructed which shows the possible execution paths through the program. When a valid pragma is parsed in the WCET module, the information

[5]Valid means a syntactically correct written and not multiple defined pragma

must be sent to the global optimizer so that infeasible paths can be removed and extra timers can be added. This is done by using the double linked list shown in fig 5:

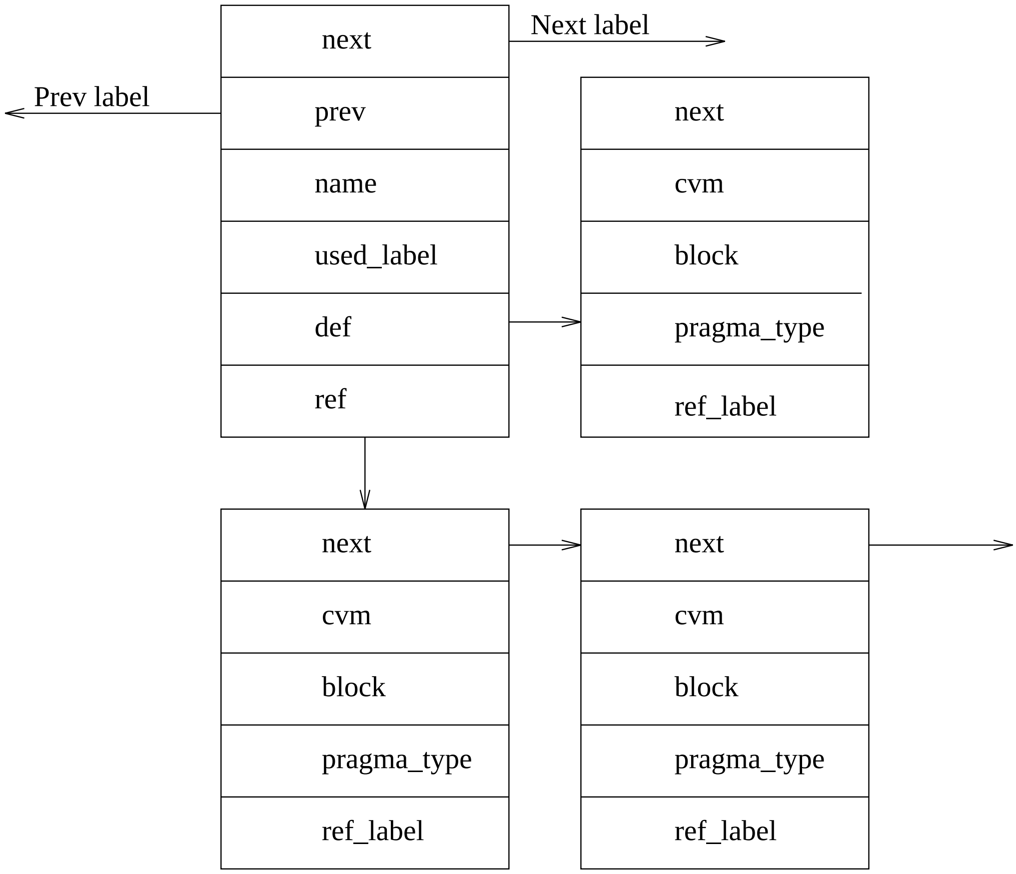

Figure 5: One label struct and three reference structs

```
typedef struct tlabel_T00
{
  struct tlabel_T00        *next;
  struct tlabel_T00        *prev;
  char                     *name;
  struct reference_rec_T00 *def;
  struct reference_rec_T00 *ref;
  unsigned short           used_label;
} label_rec_T00;
-------------------------------------------------------
typedef struct reference_rec_T00
{
  struct reference_rec_T00 *next;
  CVM_rec_D05              *cvm;
  Block_Y10                *block;
  unsigned short           pragma_type;
  struct tlabel_T00        *ref_label;
} ref_rec_T00;
```

The figure displays two different types of structures. The first one is the *label_rec_T00* struct which is the one consisting of 6 elements, and the second is the *ref_rec_T00* struct which consists of 5 elements. The variables in the *label_rec_T00* are defined as follows:

- *next* is a pointer to the next *label_rec_T00* element in the list (equal to 0 if the element is the last in the list).
- *prev* is a pointer to the previous *label_rec_T00* element (equal to 0 if it is the first element in the list).
- *name* is a char pointer to the name of the label or the name of the timer if *start(timer_name)*- or *stop(timer_name)*-pragmas are used.
- *def* is a pointer to a *ref_rec_T00* struct. If a *label(name)*- or a *start(timer_name)*-pragma is found, this data structure will be filled in. It is otherwise set to 0.
- *ref* is also a pointer to a *ref_rec_T00* element. Whenever a label is referred in a *same_path*-, *not_same_path*-, or an *exclusive_path*-pragma this structure will be filled in with relevant data.
- *used_label* is used to give information whether a certain label is used or not. This information is used to remove labels that are of no use any longer.

It is in the *ref_rec_T00* struct where the real important information is held. The *ref_rec_T00* struct contains the following data fields:

- *next* is a pointer to the next *ref_rec_T00* struct in the list.
- *cvm* is a pointer to the corresponding statement in the source code.
- *block* is a pointer to the basic block to which the statement belongs. This is needed because the timing information given by the code generator is written into these basic block structures.
- *what_type* is a short and is used to identify the pragma type. (each pragma is also given a special number).
- *ref_label* is used as follows. If you for example have a *same_path(label1, label2)* pragma, *label1* contains a pointer to *label2* and vice versa.

When a *label*-pragma is found the definition field is set (the *def* pointer). The *next* pointer in the *ref_rec_T00* struct is set to 0 because there can only be one definition of the label. The *cvm* field points to the corresponding statement in the source code, and the *block* pointer points to the corresponding basic block. Whenever a reference to this label is found it is concatenated in the list which *ref* is pointing to. If on the other hand a reference to a label that is not yet defined is found, a *label_rec_T00* struct is constructed anyway with the reference input in the *ref* list but with the *def* pointer set to 0. This means that the label is not defined yet, but may be found later in the source code. When the label is found, the *def* data field is filled in (or actually the data structure that *def* is pointing to). This list with all labels defined (which is a global list called *global_info*) and all references to these labels is what the global optimizer and the timing tool need to remove infeasible paths from the code.

There is a minor difference between the timing pragmas (*start(timer_name)* and *stop(timer_name)*) and the path-pragmas. Whenever a *start* pragma is found a *label_rec_T00* struct is constructed and the *name* field is filled in with the argument (the name of the timer). The structure that *def* points to is also filled in. When the associated stop pragma is found, the *ref* struct is filled with information (a reference). The aim is to have pointers from all references to the associated statements in the source code.

5.4 The Traversal

The code generator in the compiler calculates the execution time for each basic block. By storing this information into the corresponding basic block structure, and consequently allowing the global optimizer to take part of this information, the global optimizer has all the information needed to calculate the WCET. It is now time to traverse the flow graph and use the information given by the pragmas as well as the information concerning the execution times given by the code generator. Because there are in general a large number of possible paths through the source code, the traversal must be recursive.

Problems that have to be solved are the way timing information should be kept and how the program should be traversed. We decided to start the traversal from the end and traverse backwards to the start of the function. One reason for this is that you will in the end get all timing information in one single node, the start node (start of the function). The data structure used to keep the timing information is displayed in fig 6.

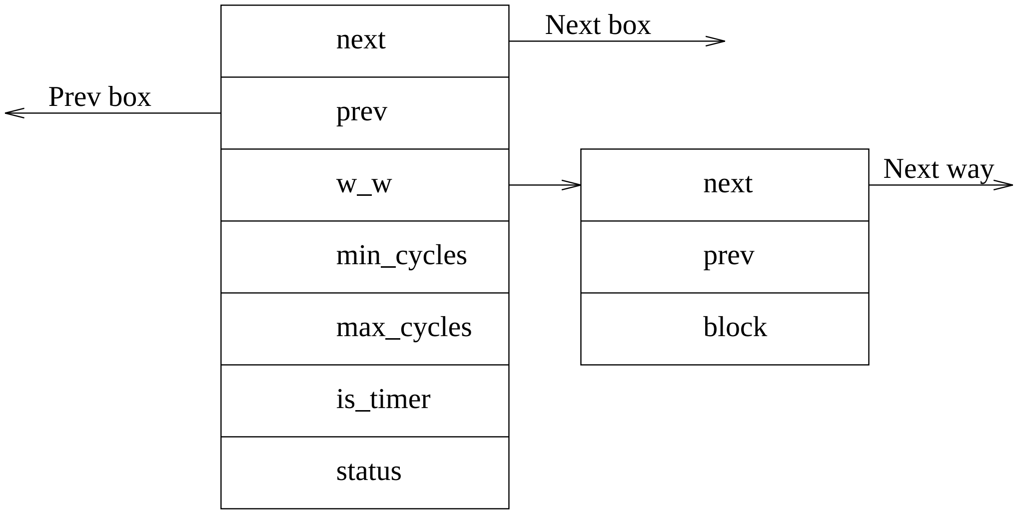

Figure 6: A timing box with corresponding "way-tracker"

```
typedef struct t_info_Y10
{
  struct t_info_Y10        *next;
  struct t_info_Y10        *prev;
  struct way_Y10           *w_w;
  unsigned long            min_cycles;
  unsigned long            max_cycles;
  unsigned short           is_timer;
  unsigned short           status;
} timing_rec_Y10;

----------------------------------------------------------

typedef struct way_Y10
{
  struct way_Y10           *next;
  struct way_Y10           *prev;
  struct block_Y10         *block;
} passed_label_rec_Y10;
```

The *timing_rec_Y10* keeps the timing information for an individual path and the passed_label_rec_Y10 keeps information on which basic blocks with a corresponding label are passed. The reason why you have to be aware of which block you have passed is for being able to remove infeasible paths from the WCET calculation. The variables in the *timing_rec_T00* struct are defined as follows:

- *next* is a pointer to the next *timing_rec_Y10* struct or 0 when the last element in the list.
- *prev* is a pointer to the previous *timing_rec_Y10* struct or 0 when the first element in the list.
- *w_w* is a pointer to a *passed_label_rec_Y10* struct which contains a pointer to passed basic blocks which have a label associated to themselves.
- *min_cycles* is an unsigned long which contains the minimum execution time (in clock-cycles) for the path.
- *max_cycles* is also an unsigned long which contains the maximum execution time (also in clock-cycles) for the path.
- *is_timer* is an unsigned short which is a flag that designates whether the struct corresponds to a timer (described in chapter 4.2.4) or not.
- *status* is also an unsigned short which designates whether the block may be removed or not. There are *timing_rec_Y10* structs that are not allowed to be removed. This will be explained later in this section.

The variables in the *passed_label_rec_Y10* struct are defined as follows:

- *next* is a pointer to the next *passed_label_rec_Y10* struct or 0 when the last element in the list.
- *prev* is a pointer to the previous *passed_label_rec_Y10* struct or 0 when the first element in the list.
- *block* is a pointer to a basic block struct. In this basic block struct there is a pointer to the corresponding *label_rec_T00* (described in figure 5) and if a timer is started or stopped from this block, there is also a pointer to that *label_rec_T00* in the basic block struct.

The traverse function takes three arguments. The first holds a pointer to a *control_flow_Y10* struct which is a structure that includes a pointer to a basic block (and some information not needed to be mentioned here). The second is an unsigned long which holds the minimum number of times the basic block may iterate (used for loops) and the third argument holds the maximum number of times the basic block may iterate. The algorithm to calculate the execution time can be summarized in 7 steps:

1. Traverse the flow graph (by recursion) until the exit of the function is found (there may be several exits but the "real" exit is the node in the "lower right corner" of the flow graph).
2. Construct a *timing_rec_Y10* struct and initialize *min_cycles* and *max_cycles* to zero. If the block has a label create a *passed_label_rec_Y10* struct and let the variable *block* point to this basic block. Add the timing cost for the basic block.
3. Export this *timing_rec_Y10* to all entries i.e. send it to all "parent basic blocks". If block B can immediately follow block A then A is a "parent basic block" to B. All parent blocks can be reached from a basic block through its entries which reside in a list of pointers to all these "parent basic blocks".

4. If the block is not an exit node for the function do the following: The block has received *timing_rec_Y10* structs (in the following we call them boxes) from all its children (may be reached by the list *exits*). This list with boxes is copied. If the basic block has got a label look at all the references to that label and do the following:

   - If a reference is a *same_path(label1, label2)* pragma check if the block with the corresponding label label2 is passed by any of the imported boxes. This can be checked by verifying the *w_w->block->label_ptr*. If label2 is found in any of the boxes remove all boxes not passing the "label2 basic block" because if the basic block with label1 is passed it is known that basic block with label2 also must be passed.
   - If a *not_same_path(label1, label2)* is found check if the basic block corresponding to label2 is passed by any box and if so remove this box since that path can never be executed at runtime anyway.
   - If an *exclusive_path(label1, label2)* is found do the same as for a *not_same_path* pragma. When the start of the function is reached, all boxes most be verified whether they include exact one of the two labels given by all *exclusive_path* pragmas.
   - If a *stop(timer_name)* pragma is found a new box is added with *max_time* and *min_time* set to zero and the *is_timer* set to TIMER. If a *start(timer_name)* pragma is found the *status* variable in the box is set to DO_NOT_REMOVE. When *status* is set to DO_NOT_REMOVE the *min_cycle* and *max_cycles* will not be incremented with the cost of parent basic blocks since then is the timer not measuring the execution time between the *start(timer_name)* and *stop(timer_name)*.

5. For the remaining boxes add the cost for the present basic block.

6. If there are more than two boxes that have not passed basic block with label (the *w_w* is pointing to 0) remove all these boxes except the one with maximum *max_cycles* and the one with minimum *min_cycles*. Thereafter export to all entries.

7. When the start of the function is reached check all the boxes for WCET and MCET (Min Case Execution Time) and for boxes that correspond to timers. Save these results and present them to the user.

In other words, the tool locates the "last exit" of the functions and creates "boxes" to save the execution times. These boxes are sent to all "parent basic blocks". The tool removes infeasible paths as defined by the pragmas and continues to send the "boxes" toward the start of the function. When the starting point is found all infeasible paths according to *exclusive_path* pragmas (i.e. paths that do not pass any of the two basic blocks given by the *exclusive_path(label1, label2)*) are removed. MCET/WCET is found by checking the "boxes" and present them together with the execution times for all timers defined.

As an illustration look at the example in fig 7, which is the same example as in chapter 4.1 . It shows the flow graph for two *if*-statements (the code given on top of next page). Both *if*-statements *if* and *else* parts are given labels (if1, else1, if2 and else2) and the execution times for the blocks are given in clock cycles. The execution cost for the conditions are included in the cost for the *if* end *else* part for the two *if*-statements. In chapter 4.1 we found that the WCET without pragmas was 200 clock cycles (100+100). We also found out that the path passing both the *if* part of first *if*-statement and the *else* part of the second *if*-statement never can be executed if the variable **a** does not change its value between the two conditions. Let us see what happens if a *same_path(if1, if2)* pragma is inserted. In fig. 8 the two basic blocks with labels *if2* and *else2* have created timing boxes, which include the execution time and the label of the basic block. These timing boxes are exported to the second condition (*if (a>5)*), see fig. 9. In fig. 10 the timing boxes are exported to the basic block with the labels *if1* and *else1*. One of the timing boxes is over-crossed because of the *same_path(if1,if2)* pragma (the path passing both *if1* and *else2* can never be executed at runtime). In fig. 11 the timing cost for the basic blocks *if1* and *else1* are added. These timing boxes are exported to the first condition (*if (a>10)*), as illustrated in fig. 12.

```
T_info same_path(if1, if2)
if (a>10)
{
  T_info label(if1)
  code that take 100 clock-cycles to execute
}
else
{
  T_info label(else1)
  code that take 25 clock-cycles to execute
}

if (a>5)
{
  T_info label(if2)
  code that take 25 clock-cycles to execute
}
else
{
  T_label label(else2)
  code that take 100 clock-cycles to execute
}
```

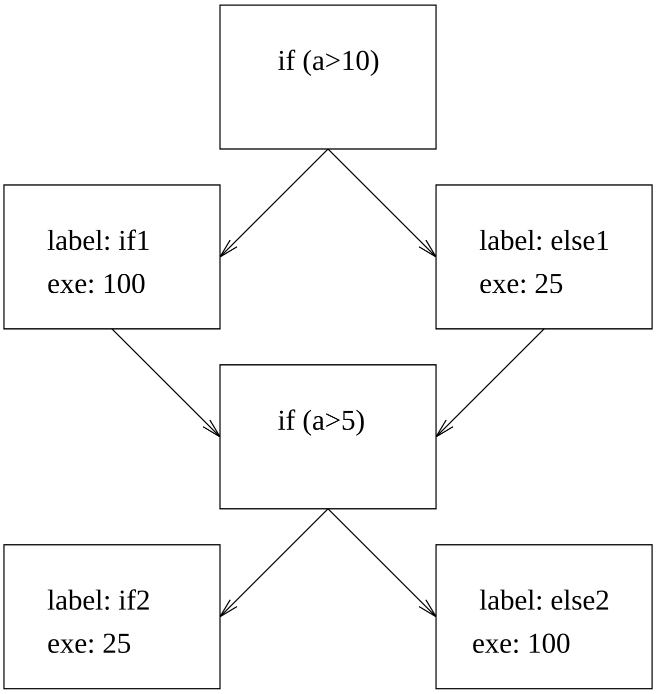

Figure 7: A flow graph for the two *if*-statements above

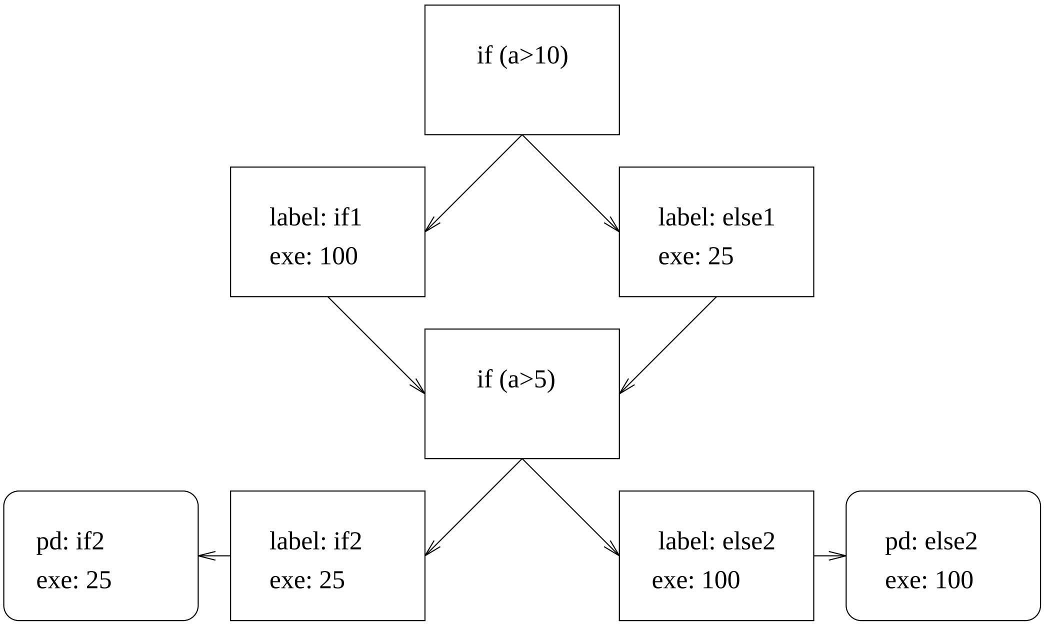

Figure 8: A flow graph with "timing boxes" on the *if* and *else* part of the second *if*-statement. *pd* stands for passed labels and the *exe* is the execution time in clock cycles.

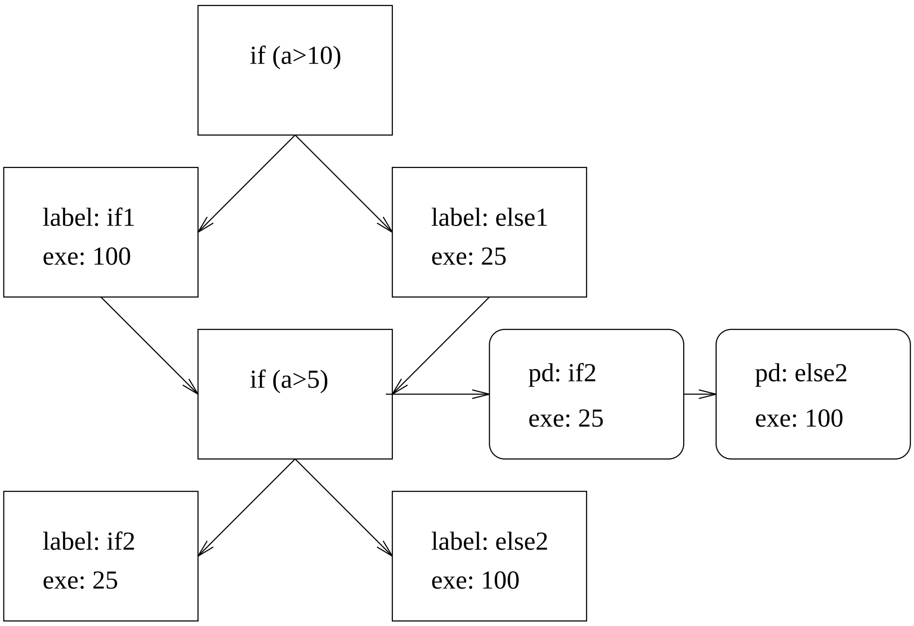

Figure 9: The "timing boxes" are exported to the second condition (second *if*-statement).

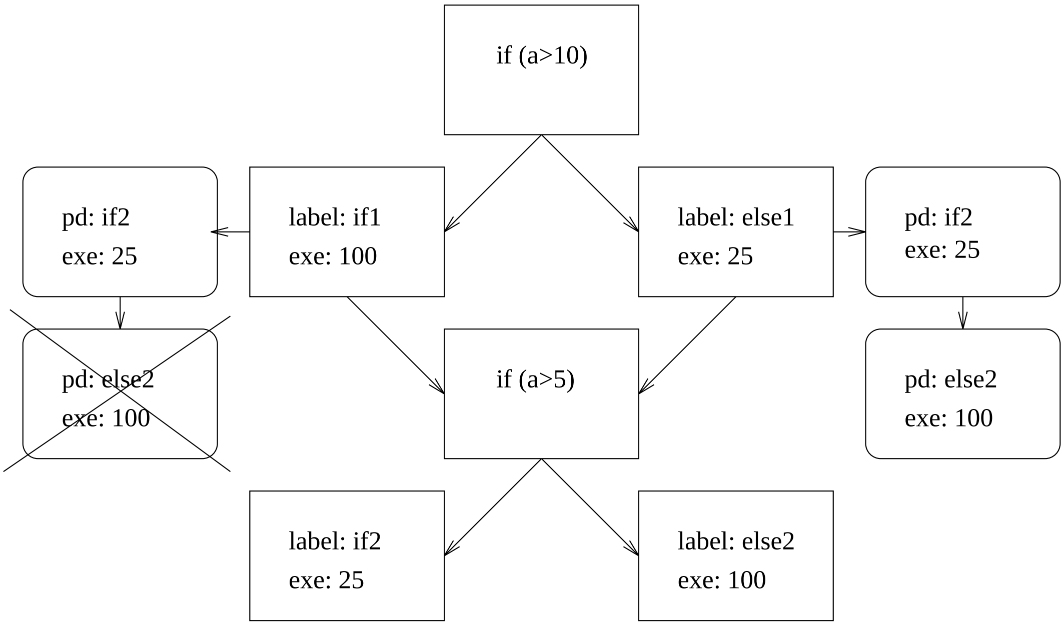

Figure 10: The "timing boxes" are exported to the *if* and *else* part of the first *if*-statement. One of the "timing boxes" are removed (over-crossed) because the *same_path(if1, if2)* pragma.

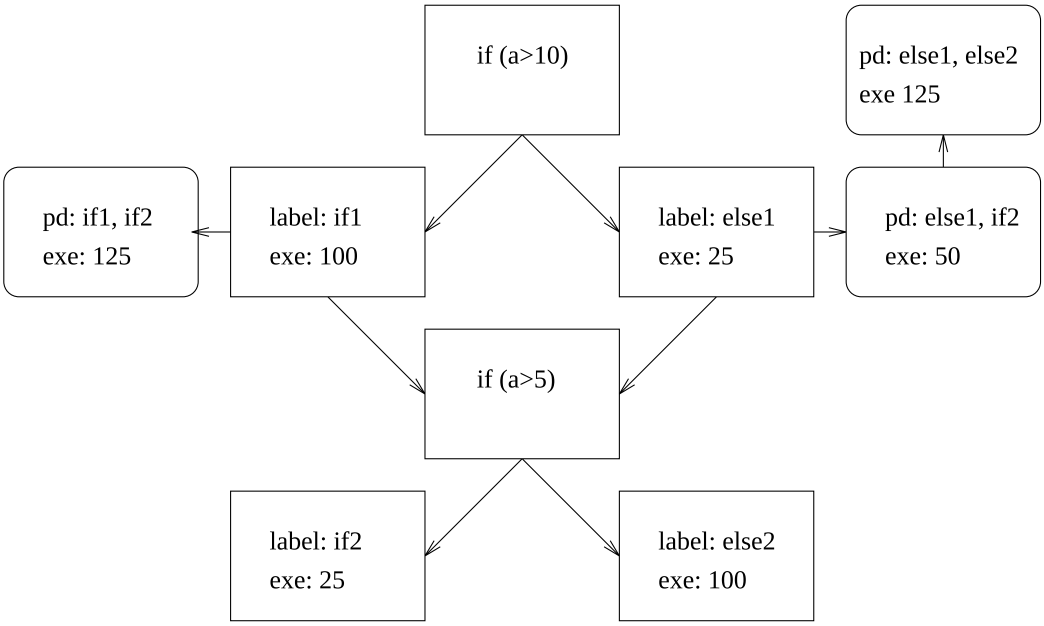

Figure 11: The timing cost for the blocks are added for the remaining "timing boxes".

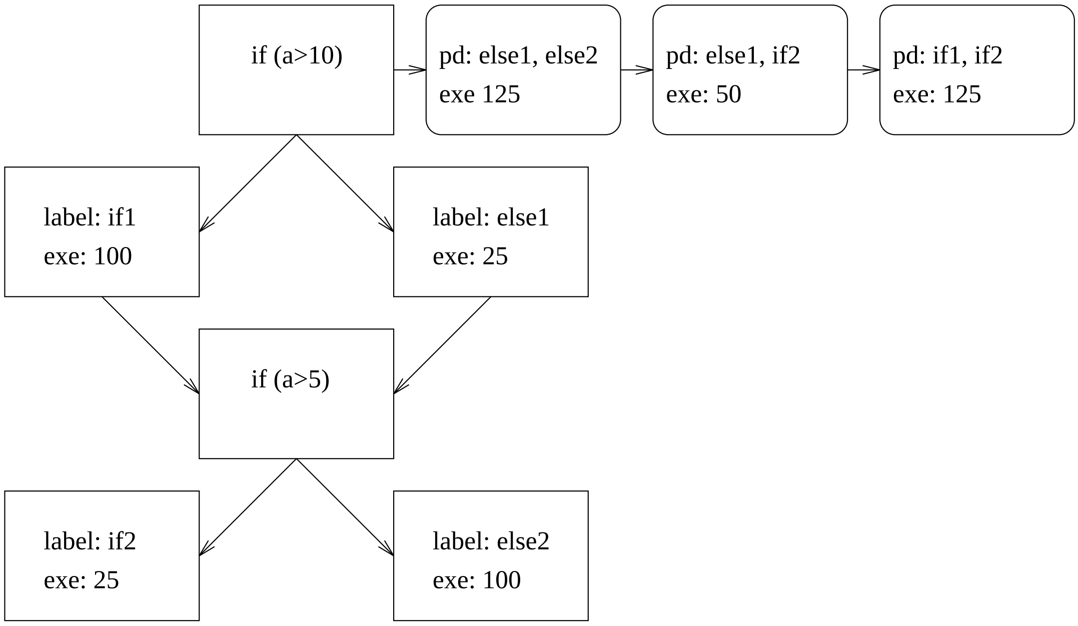

Figure 12: The "timing boxes" are exported to the first condition.

The path taking 200 clock cycles (passing if1 and else2) is removed by the tool and the WCET is 125 clock cycles (passing if1 and if2 or passing else1 and if2). By introducing the *same_path(if1, if2)* pragma, a saving of 40 % (for this example) of the WCET can be obtained. The same result can be obtained if a *not_same_path(if1, else2)* pragma was used instead of the *same_path(if1, if2)* pragma.

6 Tests and Results

Unfortunately the tool is not finished yet. Some parts in the compiler are not yet implemented and cannot be implemented by me (since they must be implemented by a pro..). The staff at IAR-Systems have too much to do so the final part of the project is postponed. There may also be errors in the code I have written but it is difficult to find out without executing the whole tool (including the "IAR"-parts).

6.1 Simulation of the Compiler

The tool has not been tested with a real compiler, instead a test of the algorithm has been done. A flow-graph was constructed (by hand) with 8 basic blocks which simulated a very small program, see fig 13. Each box corresponds to a basic block and the execution time for the blocks are given (Exe:). All basic blocks are also assigned a *label* (*label1-label8*). The program was run with and without *path*-pragmas (defined in chapter 4.2.3) and the results presented in table 3 were obtained, where the "Real WCET" is the WCET execution time found by inspecting the flow-graph.

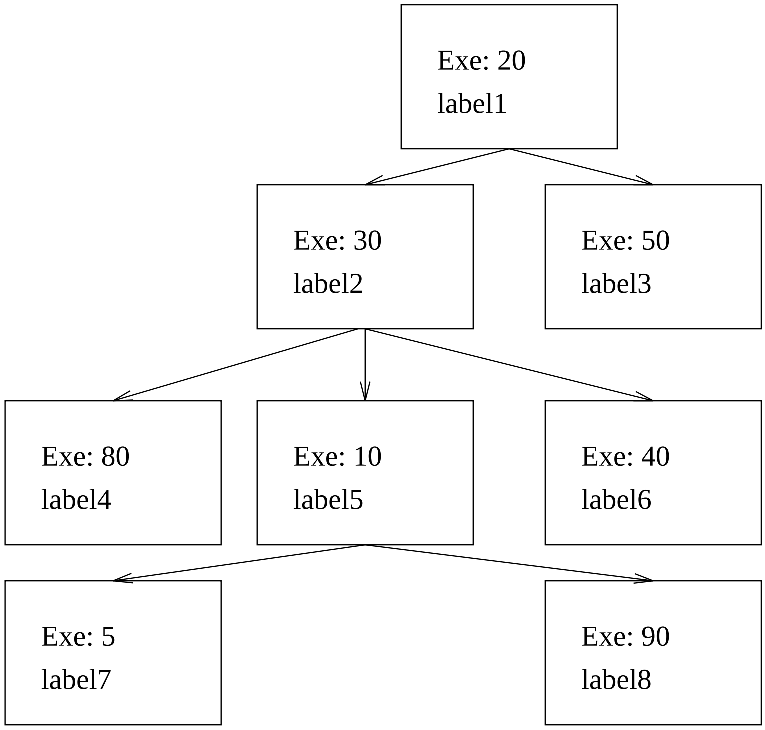

Figure 13: Flow graph for the test program

Pragma	Real WCET	Real MCET	WCET	MCET
same_path(label2, label6)	90	70	90	70
not_same_path(label2, label8)	130	65	130	65

Table 3: Results of a test run of WCET-tool

As one can see in the table, the WCET-tool is calculating the right MCET/WCET with the given pragmas.

Part II

Description of Existing Timing Tools

7 Empirical Timing Tools

There are two broad categories of timing prediction schemes: empirical timing predictions and analytic timing predictions [NR94]. The first one derives the precise timing of a task by running it with different data, and consequently measuring the execution times. One advantage with this approach is that the prediction is valid for the underlying hardware. One disadvantage is that the data set used might not be representative for the actual runtime environment. Analytic predictions calculate the execution times without having to run the program. The advantage of the analytical approach is that it is independent of the test data. Another advantage of the analytical analysis is that you are assured of getting the worst case execution time (if the underlying assumptions and formulas are correct). When using empirical timing prediction it is very difficult to know if the input data used really causes the program to run the longest (and most time consuming) path of the program. There is a possibility that the else part (for example) in an if statement is never executed with the used data. This can mean that the longest path might be missed. In larger programs it is almost impossible to run through all possible paths. *Case*-statements (*switch*-statements if you are a C hacker) and *if* statements, to name a few, lead to many possible paths of execution through the program. Work has been done [Par93]to try to decrease the number of feasible paths in a program. This means that if some path can never be executed, these should not be taken into account when worst case execution times are calculated. The problem is that it is difficult to identify these infeasible paths.

Empirical timing tools are often used as a complement to analytic timing tools. The reason is that for small programs, it is often possible to measure the WCET very close to the actual execution time, using empirical timing tools. The measured WCET can then be compared to the WCET given by an analytic timing tool to see how precise the analytic tools are calculating execution times.

7.1 The Loop Method

The loop method uses the standard system clock to measure execution times. Since system clocks are imprecise in general, we insert the code segment we want to measure in a loop. Before and after the loop a call reading the value of the system clock is inserted. This will give the time to execute X times of the code segment + X times of the loop construct. To get the "real" time, you will have to run the exact same loop, but with empty body. By subtracting this time and taking an average over the loop-count, a measure of the execution time will be obtained. It is not unusual to loop 100k times.

There are some problems with this approach.

1. If you use cache memory, there is a possibility that the empty loop fits in the cache, but the loop with the code segment does not. Since cache memory is much faster than regular memory, this will lead to an inaccurate measurement.

2. For this to work you assume that textually equivalent code requires the same amount of time to execute. This is *not* always the case.

Observe that this method will give an average execution time, and not necessary the worst case execution time.

8 Analytic Timing Tools

In this section, a couple of analytic timing tools are presented. One main advantage with analytic timing tools is that they are independent of the input data. The tool always (if it works properly) calculates the WCET. The main problem is to build a tool that correctly calculates the WCET when pipelines, cache memories, DMAs etc. are used. The fast progress in the computer area complicates the analysis. What is new today is old tomorrow. Whenever a new hardware component is installed to increase the speed of the system, the theory to calculate execution times has to be modified.

8.1 Table Lookup Method

To calculate the WCET of a program, the table lookup method uses the target assembly code. Adding the operation time and the time to prepare the operands gives the execution time for each individual instruction. The sum of the instructions execution times gives the total execution time. The instruction times can often be found in the user's manual.

There are several disadvantages of this approach [HBW94].

1. Some compilers do not generate assembly code.
2. It may be difficult to match the assembly code with the corresponding high-level language construct.
3. It does not give an accurate measure when you have a high degree of concurrency e.g when pipelining is used.
4. You are dependent on the accuracy of the timing measures provided by the vendor.

There are methods to overcome these problems. You can solve the first problem by disassembling the object code. The price paid is of course a new phase in the analysis stage. To avoid the second problem, you can insert markers around the target source code segment, that will help to match the high-level instructions with the corresponding assembly code instruction. One problem in the compiler is the optimizer, that may reposition or eliminate the markers [HBW94].

8.2 Object Code Analysis

This method is related to the approach given in 8.1. The approach consists of three stages: 1) Disassemble the object code. 2) Form basic blocks. 3) Construct an execution graph that shows how the blocks interact.

The disassembler reads the object file and converts each machine instruction to a convenient internal form. One reason for doing this disassembling is that there is no one-to-one mapping between object code and high level code. A one-to-many relationship can arise between high and low-level basic blocks for two reasons [Cha94]:

1. The design of high level language includes constructs that may lead to complex control-flows at the assembly-language level.
2. Compilers introduce additional control flows for their own reasons.

In the second phase the analyzer makes two passes over the code. During the first pass it builds a list of addresses. Whenever the analyzer finds an instruction that changes the control flow (branches, returns etc), it appends the address of the following instruction to the list. If the instruction was a call or branch, the target address is also appended to the list. The list is sorted and the addresses are potentially basic block headers. In the second pass the flow graph is constructed by using the information collected during the first pass. A time-table is used to calculate the execution times. The timing-table are just those printed in the used processors handbook.

8.3 "The Annotation Approach"

This approach is developed by A. K. Mok at the University of Texas. The tools described in [Mok89] supports C, but are table-driven and can easily be adapted for other block-structured languages. The idea is to *annotate* the programs. These markers, which are placed both at the C level and the assembly level, are used to get timing information. Information like how many times a loop will be executed or how often an if-statements else part is executed, is two examples of information that can be given by these annotations. You put the markers in the C code and then compile it with a modified C compiler. The annotations are transfered to the assembly code, but the user can still enhance the annotations.

A tool window is used to present the annotated assembly code. In this window you can also see the corresponding program flow graph. Each node in the graph corresponds to a basic block in the program. The execution time of the blocks are determined by a hardware simulator for the target processor. This simulator can be enhanced in the future to incorporate semantic constraints so tighter execution bounds can be achieved, by using more powerful analysis algorithms.

8.4 The Mars Approach

The **MA**intainable **R**eal-Time **S**ystem (MARS) is a system developed at the Technical University of Vienna in Austria. Guaranteed, deterministic behavior is one of the goals of the MARS project. To achieve this, they use specially designed compilers and schedulers, to get total control over the system. As one part of the MARS project an execution time analyzer was built. Some restrictions are made in order to get analyzable programs [PK89]. Loop-bounds is one example. Also recursion and *goto* statements are banned, as are parameters and pointers to functions, because execution times for these functions are difficult to predict. Loop-bounds can be given either as a maximum number of iterations (MAX_COUNT) or as a maximum time spent in the loop (MAX_TIME). An exception handler is invoked if any of these bounds are violated. The execution time of the exception is, of course, taken into account.

In table 4 you can see formulas for the calculation of the execution times given in [PK89]. The compiler calculates the execution times for these constructs by recursively applying these formulas.

The τ function is interesting. It takes as argument a simple action, statement, expression or subroutine, and returns the execution time derived from the number of machine instruction cycles. The function *maxt* calculates the upper bound for the execution times of the argument. To improve the time analysis, Puschner and Koza introduces extra information concerning the control flow.

Definition 1 (Scope) *A Scope is a part of a program's instruction code.*

Definition 2 (Markers) *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.*

The markers are used to give a maximum number of times the control flow can pass a specified position in a scope (a specified code segment). This can be very useful if nestled for-loops is

construct	MAXT		
primitive	*maxt(primitive)*	$=$	τ*(primitive)*
sequence	*maxt(sequence)*	$=$	$\sum_i maxt(construct_i)$
alternative	*maxt(alternative)*	$=$	*maxt(condition)+* $max(maxt(construct_i),$ $maxt(construct_2))$
$loop_{number}$	$maxt(loop_{head})$	$=$	*maxt(init)+maxt(condition)+ count*(maxt(body)+ maxt(condition))+ maxt(overrun_statement)*
	$maxt(loop_{time})$	$=$	*count*(maxt(body)+ maxt(condition))+ maxt(overrun_statement)*
$loop_{time}$	$maxt(loop_{time})$	$=$	*time+maxt(timeout_statement)*
subroutine	*maxt(subroutine)*	$=$	$\tau(organization) + maxt(body)$

Table 4: Formulas for the calculation of maximum execution time

used, where the number of iteration for the inner loop is changing (for example in bubble sort). In each scope you can have arbitrary many markers. Checking a marker costs 40 CPU cycles, but the extra information given by the marker will (often) decrease the worst case execution time much more.

Puschner and Koza have made some experiments were they analyzed a program with and without marker constructs and obtained the following result (see table 5).

Control-flow information	Execution time (clock cycles)
Bounded loops only	551 475 096
Bounded loops and markers	46 810 232

Table 5: Execution times with and without markers

This indicates that markers and scopes gives substantially more accurate WCET measures.

8.4.1 An Experiment Using Mars and Control Flow Information

Static analyzers usually overestimates the WCETs by a large margin. The reason for this is that they calculate execution-times for paths that never really can be executed. If the programmer can give information about the execution of tasks (that cannot easily be derived from the code structure itself), it is often possible to decrease the WCETs.

Static execution time analysis experiments have been done at the university of Vienna [PV91]. They wanted to investigate if extra information concerning execution of tasks really decreased WCETs. They compared the measured execution times for some tasks with the bounds for the execution times given by the static analyzer. A prototype MARS node was used to measure the execution times of the tasks. Preemptions of task executions by operating system and hardware activities (interrupts and DMA) were taken into account [PV91]. The MAXT (MAximum eXecution Time) analyzer gave bounds for the execution times of tasks. The analyzer was able to take timing parameters of the used hardwares, tasks control structures, and additional information about the control flow into account. This additional information was used to remove some infeasible paths. Markers, scopes, and loop sequence constructs was used to give this extra control flow information [PK89].

The MAXT was calculated by three different approaches.

1. A simulation tool. The tool outputs a list of possible execution times (not only the worst case).
2. The MAXT-analyzer. Computed WCETs according to rules described in table 4.
3. Manual computations. More semantic information was put into the analysis to give tighter execution time bounds.

8.4.2 The Experiments

In the experiments, the following sorting algorithms were used: bubble sort, merge sort, insertion sort, and an iterative version of heap sort (how these sorting algorithms work is out of the scope of this paper). The three approaches given earlier, were used to calculate the execution times for the sorting algorithms when different amount of control flow information were given [PV91].

1. "only loop bounds, no markers, scopes, or loop sequences (referred to as $maxt^{i}$ in the following figures);
2. loop bounds and obvious markers, scopes and loop sequences ($maxt^{ii}$);
3. loop bounds and exhaustive use of markers, scopes and loop sequences ($maxt^{iii}$), often only possible after detailed analysis of the algorithm;
4. all information we could extract from the code semantics ($maxt^{iv}$)."

The tasks were run with a large set of test data. The results given by the MAXT analyzer are presented in table 6. The results shows that unrealistic high execution times are obtained when

μs	$maxt^{i}$	$maxt^{ii}$	$maxt^{iii}$	$maxt^{iv}$
insert.	623.8	416.8		
bubble	731.0	445.8		
merge	2881.4	1165.8	757.2	600.4
heap	953.0	933.2		696.6

Table 6: Calc. MAXT Bounds for Different Sorting Algorithms

only loop-bounds are given to decrease the number of execution paths. When scopes and markers are introduced, tighter execution time bounds can often be achieved. There are programs when scopes and markers is not enough information to calculate tight[6] execution measures. Puschner and Vrchoticky claims that there are different types of dependencies between program paths that could not be expressed in this approach like "A code piece a is executed if, and only if a code piece b is executed", to give one example. If information like this could be incorporated in the analysis, even tighter execution time bounds may be achieved.

8.5 The MAXT Analyzer

The MAXT analyzer is a new (-93) task timing tool for MARS [PS93]. The tool computes the WCET but also gives information of how often a certain code segment is executed. This is valuable information because it gives an opportunity to optimize code segments that are executed often, which leads to shorter execution times. There are a couple of assumptions made for the approach:

[6] here tight means close to the actual WCET

- A task receives a set of messages, performs a calculation (task body) and sends a set of messages. Messages cannot be received or sent during the calculation part. No communication, synchronization, or unpredicted delays may occur in the task body.

- Tasks and procedures are written in MODULA/R, which is a language that guarantees that WCET of every code segment can be bounded.

The approach uses something called **Timing Trees**. They are data structures that contain all relevant information about timing behavior. The trees are strongly related to the syntax trees the compiler generates for a program. There are several reasons to use this representation [PS93]:

- The similarity to the syntax tree allows a mapping between the respective nodes of both data structures to be established.

- Maintaining the timing information in a data structure that reflects the structure of the source programs gives possibility to relate the source code and timing information. To play around and model execution times the programmer can set hypothetical times for all displayed program parts and observe the effects on other parts of the code.

- Since all tools of the programming environment can directly access the timing tree, the turnaround time during programming can be kept short.

The leaves of the tree represent basic blocks in the source code and the inner nodes keep track of possible execution paths and other structure information. Example of information in the inner nodes are loop bounds, markers etc.

One advantage of the approach is that you can handle return- and break-statements in the code. A return-statement terminates the execution of a procedure immediately. This leads to problems because subsequent statements in the procedure are not executed. To see that this really is a problem look at fig. 14 :

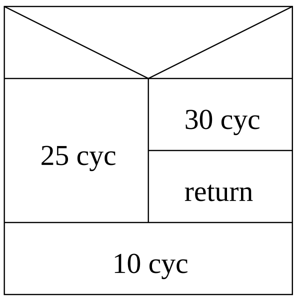

Figure 14: An if statement including a return statement

It shows an if-statement. The else part which includes one return-statement takes 30 machine cycles to execute. This is longer than the "then-branch", but is shorter than the "then-branch" plus the MAXT of the subsequent statements (25+10). This means that you will have to know the execution time for the paths that ends at the last statement of the procedure (or another return-statement), and compare this time with the execution time for the path which includes the return-statement. The solution to the problem given in [PS93] is to start calculating the MAXT from the end of the procedure instead of from the beginning.

8.5.1 More About Timing Trees

Alexander Vrchoticky explains a little more about the Timing Trees and how they are builtin his article "Compilation Support for Fine-Grained Execution Time Analysis". As previously mentioned, the Timing Trees are hierarchical representations of the block structures of procedures. To give a closer look how these Trees might look like, an example taken from [Vrc94] will be presented. Fig. 15 shows the MODULA/R code and the timing tree for a procedure gcd (greatest common divisor). In the upper right corner of the non-hidden nodes, you can see the line numbers of the corresponding source code statements. You can also see a label providing the corresponding source code line numbers in each node. One problem with constructing the Timing Trees is that all information is not available at the same time. The structure of the program and execution frequency bounds are easily collected in the first phase of the compilation, but the timing information is only available after code generation. Vrchoticky's solution to this is to create the Timing Trees in the first phase of compilation and later update it if any code-improving transformations are made, and then add the execution times after code generation.

```
1 --
2 -- Example Modula/R Procedure
3 --
4
5 procedure gcd (var a,b : 1 .. 100) : 1 .. 100;
6 var x, y : 1 .. 100;
7 begin
8
9      x := a;
10     y := b;
11
12     while x <> y max 100 times
13     do
14         if x > y then
15             x := x - y;
16         else
17             y := y - x;
18         endif;
19     endwhile
20
21     return x;
22 end gcd;
```

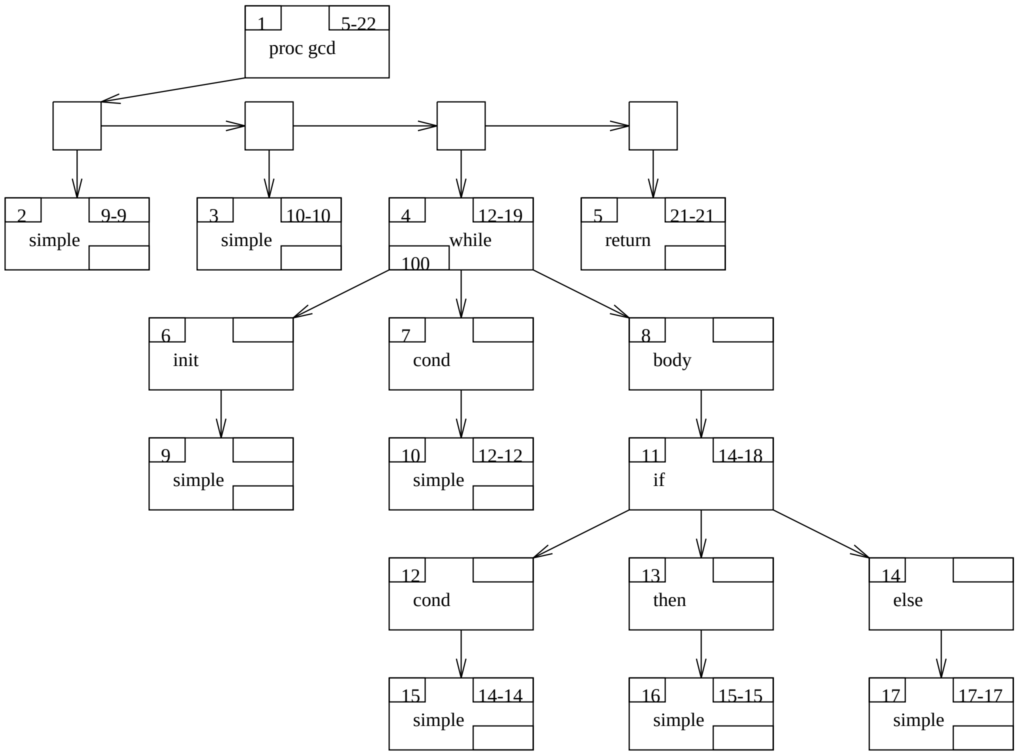

Figure 15: Code and Timing Tree for gcd procedure

8.6 Park and Shaw

C.Y. Park and A.C. Shaw [PS91, Par93] have implemented a tool that analyses a subset of C. They used the GNU C-compiler without optimization. The authors give an algorithm to get both the upper bound and a lower bound for the execution time. The approach can be summarized in the following three steps.

1. Decompose the statements into primitive component blocks.
2. Compute the times of the statements by composing the predictions of these blocks according to the system-independent language semantics. The component blocks, that are implementation dependent are predicted via object code prediction and machine analysis.
3. Applying the decomposition, prediction and composition recursively, we can compute the time bounds of a statement, a compound statement or a procedure.

The procedure gives a pair of lower and upper bounds of the execution times, T[S]=[t_{low},t_{up}] [Par93]. As an illustration consider the following example.

S1 a=b+c-d;

The instruction S1, which is composed of 7 primitive components, will give an execution time of:

T[S1]=T[b]+T[+]+T[c]+T[-]+T[d]+T[a]+T[=]

One of the reasons not to use the optimizer in the compiler is that it might use the target machine registers intelligently. If for example the variable c already is stored in a register, the optimizer will probably remove the assembly instruction **move c,d0** (something like this will be generated) and the timing analysis will be invalidated.

C.Y. Park have made some extensions [Par93] to the work described in [PS91]. In the extended work, Park is removing infeasible paths through the code by using user-provided information. This leads to tighter prediction bounds. The user provided information is given in a **I**nformation **D**escription **L**anguage (IDL). Below is a subset of the language:

- **samepath**(A,B); Statements A and B must be executed together.
- (**not** A) **imply loop** L 10 **times**; If A is not executed, then L is iterated 10 times.
- **execute** A[0,1] **times inside** L; The exception case A is executed at most once inside L.

By only considering the subset of paths satisfying the set of IDL constraints tighter execution time bounds are obtained.

8.6.1 Results

In [Par93], the author say that procedures can be split into three types. The first type are procedure whose execution times are predictable without dynamic path analysis. More than half of the tested procedures fall into this type. The second type are procedures that can be refined tight by path analysis using IDL (Information Description Language). This type is not as frequent as the first one, but can be found in procedures with many procedure calls. The third type are procedures whose predictions are loose even with path analysis. There are some results given, that shows that predictions with dynamic path analysis can give a much tighter bound on the execution time. In table 7 you can see the results when an "insertion_sort" algorithm was analyzed with and without dynamic path analysis:

This indicates that dynamic path analysis can be really valuable when calculating WCETs.

Control-flow information	Best case	Worst case
Predictions without dynamic path analysis	187.17	3450.10
Predictions with dynamic path analysis	187.17	2085.04
Measured times	197.20	2071.10

Table 7: Execution times with/without dynamic path analysis

8.7 WCET When Cycle-Stealing DMA is Used

Huang and Liu [HL95], have investigated WCET when cycle-stealing DMA is used. DMA stand for **direct memory access**. A DMA controller transfers data between the main memory and the I/O device with minimal CPU involvement. This means that the CPU can execute other instructions while the DMA controller is transferring data. There are (at least) two different kinds of DMA. One operating in burst mode and the other in cycle-stealing mode. When the DMA is working in cycle-stealing mode it “steals” bus cycles from the executing program.

Each assembler code instruction takes one or more machine cycles to execute. A machine instructions takes one or several processor cycles to execute. A “regular” instruction may look like this:

One machine cycle to fetch the instruction (bus-access).
One machine cycle to fetch the operands (bus-access).
One machine cycle to execute the instruction (no-bus-access).
One machine cycle to write the result in memory (bus-access).

In [HL95] they classify machine cycles in two classes: bus-access (B) and execution-cycles (E). The DMA controller is only allowed to access the bus when the CPU is in an E cycle. If the CPU is in a B cycle the DMA controller have to wait. To become the “bus master”, the DMA first sends a *bus request*. After this the DMA controller:

1. waits when the CPU enters a B cycle from a B cycle.
2. becomes the bus master when the CPU enters an E cycle from a B cycle,
3. continues its transfer as long as the CPU continues to be in E cycles, and
4. releases the bus when it finishes all the data transfers or when the CPU enters a B cycle from an E cycle.

The authors introduce a time called **B**us **M**aster transfer **T**ime (**BMT**), which is the time to switch the ownership of the bus between the CPU and the DMA controller. After each data unit transfer, the DMA checks if there is any pending bus request from the CPU. If so, it leaves the bus to the CPU. Otherwise the DMA controller is allowed to transfer more data if there is any. If the DMA is finished, it give away the bus to the CPU. Another time introduced is **b** which is the delay between when the CPU requests the bus and the DMA controller checks the request and releases the bus.

8.7.1 The Algorithm

If you have a sequence of machine instructions like $B_n \longrightarrow E_1 \longrightarrow E_2 \longrightarrow ... \longrightarrow E_k \longrightarrow B_{n+1}$, the execution delay of B_{n+1} is equal to (BMT+b). If you now let **m** denote the number of data transfers the DMA controller performs under the k consecutive E cycles, and **DT** be the time to do each transfer and T_{E_i} to be the execution time for machine cycle E_i. Then

$$T_k = \sum_{i=1}^{k} T_{E_i}$$

is the total execution time for the k consecutive E cycles. Huang and Liu now says that "Based on the facts that $0 \leq b \leq DT$ and $T_k + b = m * DT + BMT$, we have

$$(m-1) < \frac{T_k - BMT}{DT} \leq m$$

Therefore we conclude that

1) $$m = \lceil \frac{T_k - BMT}{DT} \rceil$$"

The CPU suffers a delay d' because of the execution of the machine cycles.

$$d' = m * DT + 2 * BMT - T_k$$

This is an upper bound because if the DMA holds the bus for less then m data transfers then there will be no delay b. The conclusion to the approach is that the exact delay suffered by the CPU execution is at most equal to

2) $$d = \lceil \frac{d'}{T_c} \rceil * T_c$$

where T_c is the clock-cycle time. The algorithm given in [HL95] to get WCET is as follows:
Input:

1. the number of CPU instructions, N, and the instructions inst[1],inst[2]....,inst[N] in the stream S.
2. the number of data transfers, M, required in the cycle-stealing DMA I/O operation.
3. the execution time of each instruction, inst[i].execution_time, and its machine cycles, for i+1,2...,N.
4. BMT and DT, the two parameters of the DMA I/O bus.

Output:

1. WCET, the worst-case execution time of the concurrent execution of the instruction stream S and the DMA I/O operation.

Procedure:

1. Set WCET and trans, the number of transfers completed, to zero.
2. For i=1 to N, compute the contribution of inst[i] to WCET and increment by the amount as follow:
   - update WCET=WCET+inst[i].execution_time.
   - if(trans<M)
     - compute the worst-case delay d suffered by inst[i] according to Eq. (2), and update WCET=WCET+d.
     - compute the number of transfers, m, completed in inst[i] according to Eq. (1), and update trans=trans+m.

3. If the I/O operation is not completed yet, increment WCET by the amount of time (M-trans)*DT to complete the last M-trans data transfer alone.

8.7.2 Restrictions for the Approach

This approach is only applicable on systems with no cache memory or pipelining. Also the number of consecutive E cycles in each instruction must be known [HL95].

8.8 The Micro-analysis Method

This method is based on using a machine description. The idea is is to use a set of translating rules that translate the compiler object code into a sequence of low level instructions. These micro-instructions is then analyzed for timing behavior via a multi-level pattern matching scheme. The Micro-analysis method can be split in three steps [HBW94]:

1. Compile the program and disassemble the object code.

2. Transform machine instructions into a sequence of primitive operations which express the functionality of each machine instruction in fine-grain detail. The transformations are performed by a parser that uses the machine description of the target processor to produce a sequence of primitive operations. This process is called *micro-translation.*

3. Scan the stream of primitive operations, to identify patterns and applying rules which either specify a replacement pattern or an execution time. The execution times are specified as integers which represent the number of clock cycles required for a pattern of primitive operations to execute. When the analysis is complete, the execution time of the target code segment is displayed as bounded integer time interval (i.e. [best case, worst case].

To give more insight in how the microinstructions look like, a subset of the micro-instructions taken from [HBW94] is presented.

ftch : instruction fetch

=%p : move program counter to Memory Address Register (MAR)

=%s : move source address to MAR

=%d : move destination address to MAR

+p : increment program counter

=sM : memory read to update source register

=dM : memory read to update destination register

=Ms : memory write operation

+sd : add source to destination

*sd : multiply source by destination

NZVCX: set condition codes

where d and s denotes the destination and source register which can not be accessed by the user.

8.8.1 Timing analysis

The timing tool consists of three parts. First is the disassembler that disassembles the object code. The second is the parser that transforms the assembler code into primitive code. The third is the timer that determines the execution time. The tool presented in [HBW94] predicts both the best and worst case execution times of the code segment. There is also a user interface that gives the user a chance to give maximum loop bounds and specify which code segment to be analyzed.

The processor used by the authors was the Motorola MC68020. It contains a three stage pipeline and always reads a long word (32 bits). Because many of the instructions only require 16 bits (1 word) this means that it is possible fetch two single word instructions at the same time. There are over 125 timing analysis rules for the MC68020. Here a couple of them are presented:

1. ftch $\longrightarrow$ PGR
2. =%p =sM $\longrightarrow$ PGR
3. =%p =dM $\longrightarrow$ PGR
4. PGR PGR $\longrightarrow$ FPR
5. FPR $\longrightarrow$ [bc,wc]

FPR means a long word (32 bits) read operation and PGR is a single word (16 bits) read operation. This means that when the timing tool finds ftch, it replaces it by a PGR. Two PGR in a row is replaced by a FPR. A FPR is replaced by bounded execution times by the rule number 5 ([best case, worst case]).

8.8.2 Results

Harmon and his staff have compared the micro-analysis method against table lookup and instruction counting methods and they found that micro-analysis out-performs[7] table lookup and instruction counting by a small margin for short programs, but this margin increases as the execution time of the program increases.

The approach has several advantages like being language and compiler independent. You can have a processor that uses pipelining. There are some disadvantages too. In this version given in [HBW94] you cannot have function calls. This might be solved in the future by using a call graph. Other problems are, of course, cache memory and DMA controllers.

8.9 WCET when Pipelined RISC Processor is Used

Pipelines are used to speed up a system. It gives an opportunity to fetch the next instruction, while the current is executing. For the calculation of execution time this leads to problems. One simple solution is to just ignore it. This will lead to overly pessimistic time-bounds, which for many applications is of no use. A couple of scientists at the Department of Computer Science, at the University of North Carolina, have presented one solution to the problem. They used a pipelined RISC processor (SPARC), with no DMA controller, virtual memory, memory refreshing, or caches. The authors approach [BJ95], is based on [PS91, Sha89], but also includes some extra theory of pipelining. The analysis is done at the assembler code level.

[7] out-performs in the sense of tighter WCET measures

8.9.1 The SPARC Processor

The SPARC architecture is unique, and needs some explanation. The register file is divided into a number of windows, which are put in a circular buffer. The windows have 32 logical registers, which are divided into four groups: eight locals, eight globals, eight "ins", and eight "outs". The windows share the eight global registers, but the local ones are, of course, private. Eight registers on each side of the window is shared with the adjacent windows. Window "i" shares the "outs" with window "i-1", and the "ins" with window "i+1". SAVE is the instruction to decrement the window number, and RESTORE is used to increment the same. The RAM memory used by the authors, had a deterministic access time of exactly one clock cycle. The processor used a four stage pipeline:

1. The fetch stage. The instruction is fetched form the memory (1 cycle).
2. The decode stage. The instruction is decoded, and the register operands are retrieved (1 cycle).
3. The execution stage. The instruction is executed (1-5 cycles).
4. The write-back stage. The result is written to the register file (1 cycle).

8.9.2 Pipeline Hazards

If the pipeline was full all the time, there would be no problem. Unfortunately there are condition when this is not true. This is due to something called **pipeline hazards**. If for example, instruction i takes two cycles to execute, the next instruction have to wait (stalled) under the second execution-cycle. This extra cycle must be added to the execution time. The authors says that "In any event, it is more logical to include the cost of the stall as part of instruction i versus instruction i+1 since it is instruction i that "causes" the stall". There are a couple of different hazards:

- Structural hazards. Occurs when the necessary hardware is not available.
- Control hazard. Occurs when the transfer of control is not known before the succeeding instruction is fetched.
- Data hazard. Occurs when one or more operands needed is not yet available.

Here is a table, taken from [BJ95], that summarize the conditions and costs for the stalls (see table 8).

# of Stalls	Previous instr / Current instr	Condition
1	load / any instr	Destination register of load instruction is a source operand of current instruction.
1	call / any instr	Register 15 (out register 7) is a source operand of current instruction
1	load-link / any instr	Destination register of load-and-link instruction is a source operand of current instruction

Table 8: Condition for stalls

8.9.3 The Algorithm

From the C-program you want to analyze, a control-flow graph is constructed. Compiling the program now gives SPARC assembler code, and the WCET for each basic block is computed. The computation is made in three steps:

1. The WCET is computed for each instruction, under the assumptions of no pipeline hazards.

2. Adjust for the three possible pipeline hazards, given in the table.

3. Summarize all the instructions in the basic blocks.

Finally there is some extra overhead, due to program startup/termination. This cost depends on the length "s" before, and the length "t" after the "execution stage", in the pipeline. The startup cost is "s" cycles, and the shutdown cost is t cycles. This extra overhead is, of course, only needed if a whole program is analyzed. In appendix A, you can find the worst case execution times for every instructions in the blocks.

8.9.4 Results

Some results are given in [BJ95]. They tested different code fragments with non-pipelined analysis, pipelined analysis, and compared these results with actual measured execution times. The code analyzed extracts the maximum value from a max-heap, and then reestablishes the heap (see table 9).

Analyzing method	Maximum
Non-pipelined analysis predicted execution times	437
Pipelined analysis predicted execution times	145
Actual execution times	121

Table 9: Results

These results shows that pipelined analysis may outperform the non-pipelined analysis by a factor three. This depends of course on the program analyzed.

8.9.5 The Pipeline Simulator Compiler

Another interesting tool is the Pipeline Simulator Compiler (psc). It is a tool that generates an execution time analyzer for any RISC processor. It takes structural, control, and data hazards into account. One of the major shortcomings of all existing execution time analyzers is that they are machine dependent. This is a serious problem because machine architectures evolve rapidly [NN94]. The interesting thing with this approach is that you give an Architecture Description File (adf) to the psc-tool, and gets an execution time analyzer as output. This means that you do not have to write a new tool, when a new RISC processor is released. You just have to fix a new architecture description file (input file). The adf consists of four sections [NN94]:

1. Macro Definition Section

2. Resource Definition Section

3. Instruction Definition Section

4. Assembly Syntax Section

The first, second and third section, are used to model the target architecture. The fourth section is used to describe a parser for the targets assembly language . The resource definition section describes the resources that are characteristic for a particular architecture, like registers and available pipeline stages etc. The instruction definition section describes whish instructions that are supported by the architecture, and which resources they require. The macro definition section gives the opportunity to define macros (which are expanded automatically). The assembly syntax section describes the syntax of the assembly language for the target computer architecture. It is a yacc-like tool (for more info about yacc see [AUS86]) that generates a parser for the assembly language. The output of the psc, the pipeline simulator, or analyzer, can now calculate the execution time of code segments. The analyzer require two inputs: the assembly code of a task, and a sequence of labels representing the sequence of instructions to be simulated. Each of these labels represent a basic block. The labels are inserted by hand, but the authors anticipate that a tool might automate the insertion of labels in future systems.

8.10 The SPARK Ada Approach

At the University of York, a couple of scientists have worked with timing analysis for SPARK Ada, which is a subset of Ada83 [CBW94, BCW95]. They have developed two approaches to calculate WCETs. One is a low-level timing tool and the second is a high-level timing tool.

In the source code the programmer may insert annotations, such as loop invariant assertions. All loops must have at least one assertion. These assertions are in addition to pre- and post-conditions so called cutpoints in the program. Because every loop have at least one assertion, means that any program can be split up in a finite set of basic paths [CBW94]. This means that a program may be modeled as a direct flow graph with the cutpoints forming the nodes and the basic paths the edges.

8.10.1 The Low-level Timing Tool

The timing tool consists of two stages:

1. The disassembling stage, where the object code is disassembled.

2. The WCET stage, where the timing info for each basic path is generated.

The disassembler annotates the code with extra timing information, and build timing trees for each subprogram body. The nodes in the trees are of the following type (see table 10) [CBW94].

Node type	Description	Attributes
basic	A basic block	WCET
oneof	Root of a tree of alternatives	
alt	Root of a tree representing a single alternative within a **oneof** tree	
excall	External subroutine call	External link name, Ada unit name
loop	Root of a tree forming a loop	
imploop	Root of a tree forming a compiler-generated loop	Loop bound
condexit	Marks the position of a conditional loop exit	Pointers to segments for the exit taken or not taken
exit	Marks the position of an unconditional loop exit	Pointer to segment following loop
bodyend	Marks the end of the code for the body of an **alt** or **loop** subtree	Pointer to next segment in control-flow order
assertion	Marks position of an assertion	
unit	Root of subtree for entire Ada unit	Unit name

Table 10: Types of nodes

The timing tree is constructed by the second tool. The timing analysis is made by a two-stage algorithm. The first stage add some **bodyend** nodes to give extra information of the control-flow order of the program. Consecutive basic nodes are "smashed" together into a single node. The compiler may generate loops that do not appear in the Ada code (**imploop**). These loops are resolved. The second stage in the algorithm traverses the timing tree. A special counter keeps track of the execution times, and when a branch is reached, all but one of the branches are pushed on the stack. The execution time is reported when an assertion node is reached, and a new path is popped from the stack. All paths from a particular assertion are analyzed when the stack is empty. Then the next assertion point is analyzed, until only the postcondition remains.

8.10.2 The High-level Timing Tool

This tool use symbolic execution. Symbolic execution is a simulated execution of a program. Unknown symbols is used to represent program variables, instead of concrete values. When a symbol is assigned a new value it is given an algebraic expression which give its value in terms of itself and other symbols. It is now possible to accurately reflect the execution times even when it depends on the input parameters.

The algorithm works like this:

- **input:** A flow graph **G** represented as a record **(N,E,s,t)** where
  - **N** is the nodes representing assertions in the subprogram.
  - **E** is the set of edges representing the paths.
  - **s** is the precondition (start node) of the subprogram.

– **t** is the postcondition (terminal node) of the subprogram.

- **Output:** A symbolic formula WCET=F(a,b,c,...) which reflect the WCET in terms of WCETs of the basic paths a,b,c and the operators +, * and max.

To give extra information, the nodes and edges of **G** is annotated with extra information [CBW94]:

Graph node attributes	
Assertion	The s-expression representing the assertion at the node.
Lexical depth	The lexical nesting depth of the node within the source code. Pre- and post-conditions both have a lexical depth of 1.
preds	A list of the predecessor edges of this node.
succs	A list of the successor edges of this node.

Graph edge attributes	
Start	The node at which this edge starts.
End	The node at which this edge ends.
PTC	The s-expression representing the path traversal condition for this path.
VC	The s-expression representing the verification condition for this path.
WCET	An s-expression representing the WCET for this path.
edge_class	One of forward, back-inner, or back-outer (see below).
lex_delta	The change in lexical depth achieved when this path is traversed.
	(For back edges only) - a symbol representing the bound on the loop formed by this edge.

Table 11: Annotations for the flow graph

The algorithm to calculate the WCET can be divided into three stages:

1. The annotation stage. Loop bounds are inserted. In the model described in [CBW94] they found it more natural to consider the bounds to be on the back edge of the graph. Ordinary loop-count annotations are therefor changed to the number of times the back edge may be traversed. This stage is only performed once but the following stages is applied once for each mode [8].

2. Dead code detection. Paths that never can be executed is removed from the analysis. These paths are said to be "dead paths".

3. The WCET generation. By reduction of the flow graph (how these reduction is made is not mentioned in the report) it is possible to only have the start and the terminal nodes s and t with a single edge between them. The resulting formula for the WCET is expressed with the execution times for the basic paths and the operators max, + and *.

In the approach given in [BCW95], they introduce a **program state vector** X. It gives the values of all variables in the program in a particular point. The authors also introduce the following:

[8] A mode is a subset to the states described by a subprogram's precondition or loop-invariant assertions

- Path Traversal Condition (PTC), which is a Boolean predicate that guarantees execution of a particular path with respect to the program state vector X.
- Path Action (A), which represents the effect on the program state X of executing that path.
- Pre-condition Q and post-condition R, which are used to specify the action of a subprogram.

A path is said to be dead, which means that if the PTC is contradictory with the paths precondition, then it can not be executed. The following example shows a dead path:

$$(Q(X) \wedge PTC(X)) = FALSE$$

Information like this are used to remove infeasible paths, which in general leads to shorter execution times.

8.11 Cache Accounting Approaches

Cache memories are used to bridge the bottleneck between “fast” clock rate and “slow” main memories. Most of the approaches to calculate execution times assume that no cache is used, because it is difficult to predict cache hits. Some work has been done in the area [MWH94, AMH94]. Their approach emphasizes instruction caches, They use something called a Static Cache Simulator, to determine if a certain instruction reference will result in a cache hit or miss. Instructions are split up in four categories:

1. **Always hit**, which means that the instruction is always in cache memory.
2. **Always miss**, which means that the instructions never is in cache memory.
3. **First miss**, which means that the instruction is not in cache memory the first time, but always in cache memory for subsequent accesses.
4. **Conflict**, which means that the instruction hit/miss depends on the control flow.

Each program/task can be split up into functions. The sequence of calls between these functions is depicted in a call graph. A function can be represented by a control-flow graph. The nodes of the graph are basic blocks, and the edges denotes legal transitions between basic blocks (control flow).

8.11.1 Instruction categorization

In [MWH94] an algorithm is presented that categorizes instructions into the four categories given above. The categorization of each instruction **i**, within a basic block **b**, is based on its position in the program line **l**=$i_0..i_{n-1}$. It is also based on the abstract cache state **s**, and on the reaching state **t**.

always-miss: A cache miss is predicted if

- i=i_0: instruction i is the first reference to program line l in b and
- l $\notin$s: l is not in the abstract cache state.

always-hit: A cache hit is predicted if

- $i \in \{i_1..i_{n-1}\}$: instruction i is not the first reference to program line l in b. Or
- - i=i_0: instruction i is the first reference to program line l in b,

- l∈s: l is in the abstract cache state, and
- $\forall_{m \to c, m \neq l} m \notin$ s: no other line m (which maps into the same cache line as l) is in the abstract cache state.

first-miss: A miss on the first reference and hits for consecutive references is predicted if

- i=i_0: instruction i is the first reference to program line l in b,
- l∈s: l is in the abstract cache state,
- $\exists_{m \to c, m \neq l} m \in s$: another line m (which maps into the same cache line as l) is also in the abstract cache state, and
- $\forall_{m \to c, m \neq l} m \in s \Rightarrow m \notin t$: all other program lines m (which maps into the same cache line as l) cannot be reached anymore if they are in the abstract cache state.

conflict: A reference may result in either a cache hit or cache miss at execution time in all other cases. In Appendix B, you can see an algorithm used to calculate the abstract cache state, taken from [MWH94].

8.11.2 The Bit-Encoding Approach

Based on the algorithm, given above, to categorize instructions a bitencoding approach has been developed. An extra bit is added to the normal instruction format (fetch-from memory bit). The instruction is fetched from main memory if the (fetch-from memory) bit is set. If the bit is not set, the instruction will be fetched from cache memory. The bit is checked in parallel with the cache tags. The interested reader can find the rules, how this fetch-from memory bit is set in [MWH94].

For an uncached system, the lower bound of the execution time is

$$t_{uncached} = i * n \text{ cycles}$$

where i is the number of instructions, and n is the number of execution cycles to fetch one instruction from main memory. If a cache memory is used then the execution time will be:

$$t_{cached} = h + m * (n + 1) \text{ cycles}$$

where h and m is cache hits and misses respectively (i=h+m). For the bit-encoded approach the time will be:

$$t_{bit\_encoded} = h' + m' * (n + 1) \text{ cycles}$$

where h', and m' also mean cache hits and misses. The difference between m' and m (h' and h) is that first-misses are taken into account in the bit-encoded algorithm. This means that $h' < h$, which in turn means that:

$$t_{cached} < t_{bit\_encoded} < t_{uncached}$$

8.11.3 Results

There are some results given in [MWH94]. The authors tested different programs (different sizes), with different cache sizes (1kB-8kB), and found an improvement in execution time of a factor 3-8, when bit-encoded analysis was used instead of systems with a disabled instruction cache without any sacrifice in the predictability of WCET. They also found that larger cache improved the benefit, especially when the program was short (entire program fits in the cache memory).

Some restrictions for the approach are given in report [MWH94]: Only direct-mapped cache configuration is allowed. Recursion is not allowed, and finally indirect calls are not handled (the static simulator must be able to construct an explicit call graph).

9 Conclusions

We have in this report presented an algorithm to calculate WCET for programs written in C. Even though all parts in the tool are not implemented yet, we think that an enhanced version of this approach can be used in a commercial timing tool. There are plans to implement such a timing tool in 1996 by IAR-Systems.

This work was done for the 8051 microcontroller but can easily be ported to other target processors. The algorithm given can be used as long as there is a flow graph that does not contain jumps such as *goto*-statements. Maybe the algorithm can be sharpened in the future so it can handle wild jumps (for example *goto*-statements). As mentioned earlier the *?*-operator is a problem because the flow graph for the operator is complex (a feature of the compiler used). If a new compiler is build a different representation may be chosen so *?*-operators can be handled (by translating *?*-operators to a couple of *if*-statements).

The algorithm of annotating the source code is very similar to the algorithm given in [PS91, Par93]. The method of annotating the source code is a problem because if the programmer annotates something that is not true (maybe something contradictory) the tool will generate invalid execution times. If a commercial tool is going to be constructed much of the analysis must be done by the compiler itself. If the compiler needs more information it should ask for it. If for example a loop is found it should ask for the loop bound etc. Compilers in general are not built with execution time analysis in mind, but if a new compiler project is going to start it is possible to design the compiler so it can handle timing analysis, such as semantic analysis etc.

9.1 Future Work

The tool presented here is a framework that can be built on in the future. The basic problems have been solved like how loops are handled and how to remove infeasible paths through conditional statements. To be able to build a commercial timing tool (which requires the tool to be more user friendly) some semantic analysis must be added in the compiler so the user can be relived from some of the burden of the annotation process. If that can be accomplished the system would be safer (less risk to annotate something that is incorrect) and also more user friendly. It is probably quite a complex problem to do semantic analysis of conditional statements since you have to know where variables used in the conditions change values etc. Variable values may also depend on input data. The programmer maybe knows that an input *integer* have a value between 0 and 10, and by knowing this he can remove some paths through conditional statements. The compiler on the other hand cannot find these infeasible paths because if the *integer* have a value greater than 50 (for example) no paths can be removed and the compiler cannot know that the input value at runtime always will be between 0 and 10.

One thing that can be added in the future is the way nestled loops where the loopbound for the inner loop is changing are handled. This can be solved by the technique explained in chapter 8.4 (scopes and markers). Other things that might be added in the future are the way cache memories and DMA controllers are handled. To include scopes and markers into the timing tool is probably not so difficult but the handling of cache memories, DMA controllers and pipelines are more complex problems.

The best approach is probably to build a new compiler and have execution time analysis in mind. The compiler should be capable to do some semantic analysis, for example check if two *if*-statements can be executed together or not and also find out the loopbounds for constant loops (loops that iterate a constant number of times).

In some applications the MCET/WCET is of no interest, instead the average case execution time (ACET) is crucial. A tool that can calculate the ACET may be interesting for several applications.

A Execution Stage Times

Name	Operation	Cycles
LDSB	Load Signed Byte (from I/O space)	2 (3)
LDSH	Load Signed Half-word (from I/O space)	2 (3)
LDUB	Load unsigned byte (from I/O space)	2 (3)
LDUH	Load unsigned halfword (from I/O space)	2 (3)
LD	Load word (from I/O space)	2 (3)
LDD	Load double word (from I/O space)	3 (4)
STB	Store byte (to I/O space)	3 (4)
STH	Store Halfword (to I/O space)	3 (4)
ST	Store word (to I/O space)	3 (4)
STD	Store double word (to I/O space)	4 (5)
ADD (ADDcc)	Add (and modify icc)	1
ADDX (ADDXcc)	Add with carry (an modify icc)	1
SUB (SUBcc)	Subtract (and modify icc)	1
SUBX (SUBXcc)	Subtract with carry (and modify icc)	1
MULScc	Multiply step and modify icc	1
AND (ANDcc)	And (and modify icc)	1
ANDN (ANDNcc)	And Not (and modify icc)	1
OR (ORcc)	Or (and modify icc)	1
ORN (ORNcc)	Or Not (and modify icc)	1
XOR (XORcc)	Exclusive Or (and modify icc)	1
XORN (XORNcc)	Exclusive Or Not (and modify icc)	1
SLL	Shift left logical	1
SRL	Shift right logical	1
SRA	Shift right arithmetic	1
SETHI	Set high 22 bits of register	1
SAVE	Save caller s window	1
RESTORE	Restore caller s window	1
Bicc	Branch on integer condition codes	1
CALL	Call	1
JMPL / RET	Jump and Link	2
RETT	Return from trap	2
Ticc	Trap on integer condition codes	1 (4 if taken)

B Algorithm to Calculate Abstract Cache States

```
input_state(top) := all invalid lines;
WHILE any changes DO
   FOR each instance of a basic block B in the program DO
      input_state(B) := NULL;
      FOR each immediate predecessor P of B DO
         input_state(B) += output_state(P)
      END FOR;
      output_state(B) := (input_state(B) +
         prog_lines(B)) - conf_lines(B)
   END FOR
END WHILE
```

C Algorithms and Data Structures

In this report the data structures used for the WCET-tool for the 8051 microcontroller are presented.

C.1 A Couple of #define Definitions

To be able to separate all pragmas a flag is used (unsigned short) which is set to the following:

```
#define LOOP_BOUND 1
#define SAME_PATH 2
#define NOT_SAME_PATH 3
#define EXCLUSIVE_PATH 4
#define LABEL 5
#define START 6
#define STOP 7
```

depending on which pragma it is. Also a couple of other "#defines" are also declared.

```
#define MAX_EXECUTION_TIME 0
#define MIN_EXECUTION_TIME 1
#define TIMER_EXECUTION_TIME 2
#define NOT_USED_LABEL 0
#define USED_LABEL 1
#define NO_TIMER 0
#define TIMER 1
#define MAY_BE_REMOVED 0
#define DO_NOT_REMOVE 1
```

What these are used for will be presented in chapter C.2.

C.2 Data structures

C.2.1 The Pragma Struct

All pragmas are defined in a global structure called *array[]* which is a vector of *id_rec_T00* elements. The *id_rec_T00* struct contains the following items:

```
typedef struct id_record_T00
{
  char                        *name;
  unsigned short              args;
  fp_T00                      *func;
  unsigned short              nr;
} id_rec_T00;
```

where

- *name* is a char pointer pointing to the name of the pragma.
- *args* is the number of arguments for the pragma.
- *func* is a pointer to a unique parsing function for the pragma. The *fp_T00* is defined like *typedef CVM_rec_D05 *fp_T00(char*, char*)* which means a function that takes two char pointers as arguments and returns a *CVM_rec_D05* pointer.

- *nr* is the number of letters in the pragma name.

This structure make parsing pragmas quite easy and it makes it also easy to add more pragmas in the future by just adding an element in the vector *array[]*.

C.2.2 The Label Struct

To be able to give the information given by the pragmas to the global optimizer (used later during the traversal) a special data structure *label_rec_T00* was created. The *label_rec_T00* contains the following objects:

```
typedef struct tlabel_T00
{
  struct tlabel_T00          *next;
  struct tlabel_T00          *prev;
  char                       *name;
  struct reference_rec_T00   *def;
  struct reference_rec_T00   *ref;
  unsigned short             used_label;
} label_rec_T00;

------------------------------------------------------

typedef struct reference_rec_T00
{
  struct reference_rec_T00   *next;
  CVM_rec_D05                *cvm;
  block_Y10                  *block;
  unsigned short             pragma_type;
  struct tlabel_T00          *ref_label;
} ref_rec_T00;
```

where

- *next* is a pointer to the next *label_rec_T00* element in the list (equal to 0 if the element is the last in the list).
- *prev* is a pointer to the previous *label_rec_T00* element (equal to 0 if it is the first element in the list.
- *name* is a char pointer to the name of the label or the name of the timer if *start(name)*- and *stop(name)*-pragmas are used.
- *def* is a pointer to a *reference_rec_T00*. If a *label(name)*- or a *start(name)*-pragma is found, this data structure will be filled in. It is otherwise set to 0.
- *ref* is also a pointer to a *reference_rec_T00*. Whenever a label is referred to in a *same_path*-, *not_same_path*-, *exclusive_path*- or an *stop(name)*-pragma this data structure will be filled in with information.
- *used_label* is used to give information if a certain label is used or not. This information is used to remove labels that are of no use any longer. If the label is not used yet *used_label* is set to NOT_USED_LABEL as defined in chapter C.1. If it is used, *used_label* is set to USED_LABEL.

The *ref_rec_T00* contains the following data fields:

- *next* is a pointer to the next *ref_rec_T00* in the list.
- *cvm* is a pointer to the corresponding statement (or actually a pointer to a CVM_rec_D05 structure) in the source code.
- *block* is a pointer to the basic block which the statement belongs to. This is needed because the timing information given by the code generator is written into this basic block structures.
- *what_type* is a short and is used as an identifier of which pragma it is (each pragma is also given a special number) as defined in chapter C.1.
- *ref_label* is used as follows. If you for example have a *same_path(label1, label2)*-pragma, *label1* has got a pointer to *label2* and vice versa.

All the labels are kept in a global list called *global_info*.

C.2.3 The Timing Structures

To save the timing information a special data structure *timing_rec_Y10* were constructed which have the following data fields:

```
typedef struct t_info_Y10
{
  struct t_info_Y10         *next;
  struct t_info_Y10         *prev;
  struct way_Y10            *w_w;
  unsigned long             min_cycles;
  unsigned long             max_cycles;
  unsigned short            is_timer;
  unsigned short            status;
} timing_rec_Y10;

----------------------------------------------------------

typedef struct way_Y10
{
  struct way_Y10            *next;
  struct way_Y10            *prev;
  struct block_Y10          *block;
} passed_label_rec_Y10;
```

The *timing_rec_Y10* keeps the timing information for a path and the *passed_label_rec_Y10* keeps information about which basic blocks (with a corresponding label) are passed. The reason why you have to be aware of which block you have passed is to be able to remove infeasible paths from the WCET calculation due to *same_path*, *exclusive_path* or *not_same_path* pragmas. The variables in the *timing_rec_Y10* are used for:

- *next* is a pointer to the next *timing_rec_Y10* struct or 0 if it is the last element in the list.
- *prev* is a pointer to the previous *timing_rec_Y10* struct or 0 if it is the first element in the list.

- *w_w* is a pointer to a *passed_label_rec_Y10* struct which is a struct containing a pointer to passed basic blocks which have a label associated to itself.

- *min_cycles* is an unsigned long which contain the minimum execution time (in clock cycles).

- *max_cycles* is also an unsigned long which contain the maximum execution time (also in clock cycles).

- *is_timer* is an unsigned short which is a flag that tells if the struct corresponds to a timer or not.

- *status* is also an unsigned short which tells if the block may be removed or not. There are *timing_rec_Y10* structs that are not allowed to be removed which will be explained later.

The variables in the *passed_label_rec_Y10* struct are used for the following:

- *next* is a pointer to the next *passed_label_rec_Y10* struct or 0 if it is the last element in the list.

- *prev* is a pointer to the previous *passed_label_rec_Y10* struct or 0 if it is the first element in the list.

- *block* is a pointer to a basic block struct. In the basic block struct there is a pointer (*label_ptr*) to the corresponding *label_rec_T00* (described in figure C.2.2) if there is a label corresponding to the block and if a timer is started or stopped from this block, there is also a pointer (*timer_ptr*) to that *label_rec_T00* in the basic block struct.

C.2.4 The "Exclusive Queue"

If *exclusive_path(label1, label2)* pragmas are used, each path must pass exactly one of the two labels *label1* and *label2*. All labels that are referenced in an *exclusive_path* are therefor put in a special list. The elements in the list are of the type *exclusive_rec_T00* and they contain the following variables:

```
typedef struct exclusive_label_T00
{
  struct exclusive_label_T00 *next;
  struct tlabel_T00          *label1;
  struct tlabel_T00          *label2;
} exclusive_rec_T00;
```

where

- *next* is a pointer to next *exclusive_rec_T00* element.

- *label1* and *label2* are pointers to the two *label_rec_T00* structures referenced by the *exclusive_path* pragma.

When a function has been traversed, this list with *exclusive_rec_T00* is used to check that all paths is actually passing one of the two labels *label1* or *label2* (for all *exclusive*-path pragmas) and if not, that path will be removed from the MCET/WCET calculation.

C.2.5 The "Result Struct"

After that all infeasible paths are removed the search for the MCET/WCET and execution times for timers can start. All paths are gone through to find the MCET/WCET (and the timers). The result are saved in a *max_min_rec_T00* struct which have the following outlook:

```
typedef struct max_min_T00
{
  struct max_min_T00 *next;
  unsigned short     max_or_min;
  struct t_info_Y10  *box;
} max_min_rec_T00;
```

where

- *next* is a pointer to the next element in the list.
- *max_or_min* is an unsigned short which is set to one of the three defined values MAX_EXECUTION_TIME, MIN_EXECUTION_TIME, or TIMER_EXECUTION_TIME (defined in chapter C.1) depending on if the *min_max_rec_T00* struct corresponds to WCET, MCET, or timer execution time.
- *box* is pointing to the *timing_rec_Y10* structure where the timing information can be found (*min_cycles* and *max_cycles*).

D Description of Functions

In this report all functions included in the WCET-tool (Worst Case Execution Time) are presented. They are put in alphabetic order of the function names. The function are presented as follows:

```
type1 function_name(type2 name1, type3 name2 ...)
```

where type1 is the return type of the function, **function_name** is the function name (no shit) and type2 and type3 is variable types for the variables name1, name2 etc.

- timing_rec_Y10 ***add_cost_for_block**(block_rec_Y10 *temp,
  timing_rec_Y10 *t_rec,
  unsigned long min,
  unsigned long max)
  - Add timing cost for a basic block. This is done for all *timing_rec_Y10* structs in the list except for *timing_rec_Y10* structs that corresponds to *timer*-pragmas where both the *start*- and *stop*-pragma is found. Returns the list with *timing_rec_Y10* structs.
- timing_rec_Y10 ***add_path**(block_rec_Y10 *block, timing_rec_Y10 *path)
  - Copy a *timing_rec_Y10* with its corresponding passed_label_rec_Y10 and also add the present label into the list of passed labels (a new *passed_label_rec_Y10* struct). Returns the copied *timing_rec_Y10* struct.
- label_rec_T00 ***check**(char *str)
  - Search for a special label with the name given by *str*. If this label is not found a new label is constructed by calling the function *get_label(str)*. Returns the searched *label_rec_T00* if found otherwise it returns a new *label_rec_T00*.

- void **check_all_exclusive_paths**(exclusive_rec_T00 *exc,
  block_rec_Y10 *block)

  – Check for all *exclusive_path(label1, label2)*-pragmas that all paths pass exactly one of the *labels* (basic blocks) *label1* or *label2*. All paths not doing this is removed from the MCET/WCET calculations.

- unsigned short **check_all_pragmas**()

  – Check *global_info* that all pragmas are defined and referenced and if not, an error will occur. Returns 1 if all pragmas are defined and referenced, else returns 0.

- timing_rec_Y10 ***check_references**(block_rec_Y10 *b)

  – Check references for a label and remove infeasible paths if possible. This is done by looking on the *same_path*, *not_same_path* and *exclusive_path* pragmas and depending on if a certain label is passed or not, infeasible paths may be removed. Call for the function *add_path*. Returns a list with *timing_rec_Y10* where infeasible paths have been removed.

- void **concat**(CVM_rec_D05 *info, label_struct_T00 *t)

  – If a reference to a certain *label_rec_T00* is found that reference is added in the *ref* queue for that *label_rec_T00*.

- timing_rec_Y10 ***create_new_box**()

  – Create a new *timing_rec_Y10* struct ("timing box") and initialize *min_cycles* and *max_cycles* to 0. Returns the new *timing_rec_Y10* struct.

- void **fill_in_label**(CVM_rec_D05 *info, label_struct_T00 *t)

  – Fill in the *def*-struct for a *label_rec_T00*. This is done if a *label* pragma or a *start* timer pragma is found.

- exclusive_rec_T00 ***find_exclusive**(block_rec_Y10 *block)

  – Search for all labels that are referenced with an *exclusive_path* pragma and put all these labels in a list. This list is later used by the function *check_all_exclusive_paths*. Returns a list with *exclusive_rec_T00* elements which include all paths including *exclusive_path* pragmas.

- void **find_max_min**(max_min_rec_T00 *min, max_min_rec_T00 * max,
  max_min_rec_T00 *timer_exe,
  timing_rec_Y10 *path)

  – Search for MCET/WCET for a function and also execution times for all defined timers (the execution time between *start(timer)* and *stop(timer)*.

- CVM_rec_D05 ***func_l_b**(char *str, char *not_used)

  – Parse a loop-bound pragma to find the min/max iteration numbers. Returns the statements corresponding *CVM_rec_D05* structure.

- CVM_rec_D05 ***func_l_s**(char *str, char *id)

- Parse a *label*, *start* or *stop* pragma. Fill in *def* struct if it is a *start* or *label* pragma (construct a *label_rec_T00* if no *label_rec_T00* is existing with the right *name*. Add an extra reference if a stop pragma is found for the corresponding *label_rec_T00*. The function calls *check*, *fill_in_label* and *put_in_label_queue*. Returns the statements corresponding *CVM_rec_D05* structure.

- CVM_rec_D05 ***func_s_p**(char *str, char *id)
  - Parse the *same_path*, *not_same_path* or *exclusive_path* pragma to find the two arguments (*label1* and *label2*) and fill in information in the *CVM_rec_D05* structure. Also add the references to the two labels in the labels *ref* lists. Call for the functions *concat*, *put_in_label_queue* and *path_ref*. Returns the statements corresponding *CVM_rec_D05* structure.

- label_struct_T00 ***get_label**(char *str)
  - Construct a new *label_rec_T00*. The *name* of the label is filled in with the name given by *str* and the *def* and *ref* field are set to 0. Returns a new *label_rec_T00*.

- CVM_rec_D05 ***parse**(char *str)
  - Check which kind of pragma the *str* is and call the right parsing function for that pragma. Call the *func_s_p*, *func_l_b* or *func_l_s* functions depending on which pragma it is. Generate an error if *str* is not matching any of the defined pragmas (defined in the *array[]* structure). Returns a *CVM_rec_D05* structure which corresponds to the statement where the pragma was found.

- void **path_ref**(label_struct_T00 *t1, label_struct_T00 *t2)
  - For a *same_path(label1, label2)* (or *not_same_path* or *exclusive_path)* there must be a connection between the *label1* and *label2*. This is done by setting the variable *ref_label* to point at the other label and vice versa.

- void **put_in_label_queue**(label_struct_T00 *temp)
  - Add a new *label_rec_T00* into the global list *global_info*.

- void **remove_used_labels**()
  - Remove labels that are used and will not be used any more in the *global_info* list.

- timing_rec_Y10 ***remove_ways**(timing_rec_Y10 *t_rec)
  - Remove all paths except two (min and max) that do not pass any labels (basic block with label). Paths that corresponds to *timer* pragmas and paths passing basic blocks with labels will not be removed. Returns a list of *timing_rec_Y10* where unimportant paths have been removed.

- void **traverse**(control_flow_Y10 *list, unsigned long min_factor,
  unsigned long max_factor)
  - Traverse the flow graph for a function recursively and build up timing boxes etc. Traverse call for the functions *create_new_box*, *check_references*, *add_cost_for_block* and *remove_ways*.

- void **wcet_calc**(control_flow_Y10 *list)

– This function is called when the execution time analyzer starts. *wcet_calc* checks all pragmas and call the function traverse. Thereafter it calls for *find_exclusive*, *check_all_exclusive_paths*, *find_max_min* and *remove_used_labels*.

References

[AMH94] R.D. Arnold, D.B. Mueller, and M.G. Harmon. Bounding worst-case instruction cache performance. *Proceedings of the IEEE Real-Time Systems Symposium*, pages 172–181, December 1994.

[AUS86] S. A. Aho, J. D. Ullman, and R. Sethi. *Compilers - Principles, Techniques and Tools.* Addison Wesley, 1986.

[BCW95] A. Burns, R. Chapman, and A. Wellings. Combining Static Worst-Case Timing Analysis and Program Proof. *Real-Time Systems*, May 1995.

[BJ95] S. J. Bharrat and K. Jeffay. Predicting Worst Case Execution Times on a Pipelined Risc Processor. Technical report, Department of Computer Science, University of North Carolina, April 1995.

[CBW94] R. Chapman, A. Burns, and A Wellings. Integrated Program Poof and Worst-Case Timing Analysis of SPARK Ada . *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[Cha94] R. Chapman. Program timing analysis. Technical report, Dependable Computer Systems Centre, University of York, May 1994.

[HBW94] M.G. Harmon, T.P. Baker, and D.B. Whalley. A Retargetable Technique for Predicting Execution Time of Code Segments. *REAL-TIME SYSTEMS*, 7(2):159–182, September 1994.

[HL95] T. Huang and J. W.-S. Liu. Predicting the Worst-Case Execution Time of the Concurrent Execution of Instructions and Cycle-Stealing DMA I/O Operations. *Workshop on Languages, Compilers and Tools for Real-Time Systems*, June 1995.

[Mok89] A. K. Mok. Evaluating Tight Execution Time Bounds of Programs by Annotations. *Prcedings of the 6th IEEE Workshop on Real-Time Operating System and Software*, May 1989.

[MWH94] F. Mueller, D. B. Whalley, and M Harmon. Predicting Instruction Cache Behavior. *Proceedingas of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[NN94] K. Narasimhan and K. D. Nilsen. Portable Execution Time Analysis for RISC Processors. Technical report, Department of Computer Science, Iowa State University, 1994.

[NR94] K. D. Nilsen and B. Rygg. Worst-Case Execution Time Analysis on Modern Processors. Technical report, Department of Computer Systems, Iowa State University, 1994.

[Par93] Y. P. Park. Predicting Program Execution Time by Analysing Static and Dynamic Program Paths. *REAL-TIME SYSTEMS*, 5(1):31–62, March 1993.

[PK89] P. Puschner and C. Koza. Calculating The Maximum Execution Time Of Real-Time Programs. *Journal of Real-Time Systems*, pages 159–176, Sept 1989.

[PS91] C.Y. Park and A.C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, pages 48–57, May 1991.

[PS93] P. Puschner and A. Schedl. A Tool for the Computation of Worst Case Task Execution Times. Technical report, Technical University of Vienna, 1993.

[PV91] P. Puschner and A Vrchoticky. An Assesment of Task Execution Time Analysis. Technical report, Technical University of Vienna, 1991.

[Sha89] A Shaw. Reasoning About Time in Higher-Level Language Softwar. *IEEE Transaction on Software Enginering*, 15(7):875–889, July 1989.

[SIE88] SIEMENS, Balanstrasse 73, D-8000 München 80. *Microcomputer Components, User's Manual 10.88*, 1988.

[Vrc94] A. Vrchoticky. Compilation Support for Fine-Grained Execution Time Analysis. Technical report, Technical University of Vienna, 1994.