Towards Industry Strength Worst-Case Execution Time Analysis

Jakob Engblom[⋆] Andreas Ermedahl[⋆] Mikael Sjödin[⋆]
Jan Gustavsson[†] Hans Hansson[†]

April, 1999

[⋆] Department of Computer Systems, Uppsala University, Sweden
http://www.docs.uu.se/,
{jakob, ebbe, mic}@docs.uu.se
Technical Report# DoCS 109
Astec Report# 99/02
[†] Mälardalen Real-Time Research Center, Sweden
http://www.mrtc.mdh.se/,
{jgn, han}@idt.mdh.se
Technical Report# MDH-MRTC–2–SE

Abstract

The industrial deployment of real-time modeling and analysis techniques such as schedulability analysis creates an urgent need for methods and tools to calculate the execution time of real-time software. Recent advances in this area are promising, but there is yet no integrated solution taking into account the multitude of aspects relevant for execution time analysis in an industrial context.

In this article we present an architectural framework for execution time analysis. The framework is modular and extendible, in that interfaces and data structures are clearly defined to allow new modules to be added and old ones to be replaced, e.g., making it possible for analysis modules for different CPUs to share the same program flow analysis modules.

The architecture is intended to be included in an integrated development environment for real-time programs thereby allowing engineers to integrate execution time analysis in their everyday edit-compile-test-debug cycle.

In addition to presenting the architecture, we demonstrate that the proposed architecture is general enough to integrate existing execution time analysis techniques.

1 Introduction

An increasing number of vehicles, appliances, power plants, etc. are controlled by computer systems interacting in real-time with their environments. Since failure of many of these real-time computer systems may endanger human life or substantial economic values, there is a high demand for development methods which minimize the risk of failure.

During the last decades *schedulability analysis* has been developed to support the development of real-time system [ABD+95, SSNB95]. Schedulability analysis is performed á priori to ensure that a set of processes running in a computer system will always meet their deadlines. Today, schedulability analysis has matured to a degree where it is practically useful in an industrial engineering context [CRTM98]. One of the key components in the software abstraction used to perform schedulability analysis is the *Worst-Case Execution Time* (WCET) of software components. Knowledge of the WCET is also useful in modeling, simulation and formal verification of time critical systems, when implementing time critical sections of code or when tuning the performance of a program.

While modern schedulability analysis dates back to the 70's [LL73], the problem of determining a program's WCET (called *WCET analysis*) remained largely untackled until the late 80's [PK89]. During the last few years WCET analysis has been studied with an increasing intensity by a number of research groups around the world. The vast majority of existing results are focused on some particular subproblem of WCET analysis, e.g. semantical analysis to determine the possible set of worst case paths [CBW94, EG97, HSRW98, PS90, PS95], or modeling of architectural features such as pipelines and caches [FMW97, HAM+99, LMW96, LBJ+95, LS98, OS97].

The advances in WCET analysis, together with the demand for WCET estimates resulting from the industrial deployment of real-time scheduling techniques, makes it both possible and appropriate to introduce WCET analysis in an industrial development context. (Not withstanding that there still are many interesting technical challenges to address.) The vision of our research group is to create a WCET tool which will be accepted and used by real-time practitioners. We are convinced that this requires WCET analysis to be integrated with program development as a natural part of the edit-compile-test-debug cycle.

A WCET tool should ideally be a component in an integrated development environment, making it a natural part of the real-time programmers' tool chest, just like profilers, hardware emulators, compilers, and source-code debuggers. In this way, WCET analysis will be introduced into the natural work-flow of the real-time software engineer. To this end we are cooperating with IAR Systems (Uppsala, Sweden), an embedded systems programming-tools vendor. IAR Systems and our focus is on embedded systems. Methods for these type of systems must allow for a multitude of different hardware variants to be analyzed, since embedded systems capabilities range from 8-bit CPUs with a few hundred bytes of memory to 32-bit RISC processors with pipelines, caches and megabytes of memory.

Today, most embedded systems are programmed in C, C++ and assembly language [SKO+96]. In the future more sophisticated languages, like Ada and Java, are likely to

be more widely used. Thus, since imperative languages are dominant in the embedded systems market, we limit ourselves to these languages, but make no assumptions about which specific language.

In this article we present an architecture for a WCET analysis tool, with the purpose of creating a framework which allows available pieces of WCET analysis to be integrated and missing pieces to be identified. The architecture should be general enough to be able to incorporate and reuse the work of other researches in the field, without sacrificing precision in the analysis for the sake of generality. The architecture should also allow less precise WCET analysis, to shorten the analysis time in cases when a less accurate estimate can be tolerated or to facilitate faster re-targeting to new hardware platforms.

To our knowledge only one other complete architecture for WCET analysis has been presented in the literature [HAM+99]. However, this architecture does not explicitly consider the integration of different methods for solving various subproblems of WCET analysis.

The main contributions of this article are the following:

- We present a general architecture for WCET analysis, incorporating both source-code oriented program-flow analysis and target-machine specific low-level analysis.
- We present the state-of-the-art WCET research results and illustrate how these can be integrated into the proposed architecture.
- We provide well-defined interfaces between the modules of our architecture. This facilitates modular extensions and upgrading of WCET tools built upon the architecture, which in turn allows qualitative comparisons between different approaches to the various subproblems of WCET analysis.

Article Outline: We start, in Section 2, with an overview of the WCET analysis process, and in Section 3 we present our architectural framework for WCET analysis. In sections 4 to 7 we present the components of the architecture in greater detail, and show how current WCET analysis techniques fit into our architecture. Section 8 outlines our future plans, and in Section 9 we give some concluding remarks.

2 Overview of WCET Analysis

The goal of WCET analysis is to generate a *safe* (i.e. no underestimation) and *tight* (i.e. small overestimation) estimate of the worst-case execution time of the program. A related problem is that of finding the *Best-Case Execution Time* (BCET) of a program. BCET values are sometimes of interest, e.g. in limiting the jitter in schedulability analysis, but will not be considered further in this paper since BCET analysis is similar to WCET analysis and since WCET is the primary concern for real-time systems. See Figure 1 for an illustration of WCET, BCET, tightness, and safe estimates.

When performing WCET estimation we assume that the program execution is uninterrupted (no preemptions or interrupts) and that there are no interfering background activities, such as direct memory access (DMA) and refresh of D-RAM. Timing interference

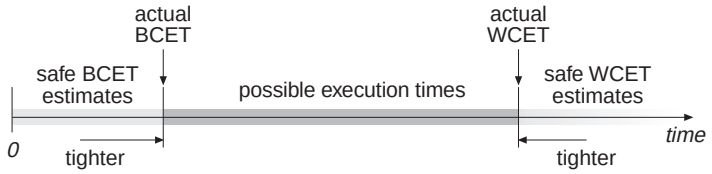


Figure 1: The relation between WCET, BCET, and possible program execution times.

caused by this type of resource contention is postponed to the subsequent schedulability analysis [BMSO+96, LHS+96].

To give an intuition for how WCET analysis is performed, we will follow the work flow of a program from source code to the final WCET estimate. The first step is to compile the source code into *object code*.

In order to determine the worst-case execution time of the program, we need to analyze the program flow. This gives us *flow information* that provides information about which functions get called, how many times loops iterate, etc. The flow analysis can either be performed in conjunction with the compilation of the program, e.g. by a special compiler module analyzing the program, or by a separate tool.

The worst-case execution time of the program depends on both the flow of the program (as reflected by the program flow information) and the object code. The object code is (conceptually) partitioned into *basic blocks*[1]. The flow information determines how to sequence the basic blocks to form the "worst-case execution" of the program.

In order to calculate the worst-case execution time, we need to determine the execution time for each basic block. Unfortunately, contemporary hardware features like pipelines, caches, and branch predictors make this quite complicated. For older micro-controllers like the Z80 [ZiL95] and 8051 [Int94], each instruction has a fixed execution time and the execution time of a basic block can be determined by simple addition. For modern CPUs, the instructions typically have a variable execution time, e.g. for a pipelined CPU, the execution time of an instruction varies with the amount of overlap with neighboring instructions. For a CPU with cache memory, the time required for memory accesses depends on whether the accessed memory word is in the cache or not. Tight WCET analysis requires these types of hardware features to be taken into account. Determining the execution time of basic blocks is known as *low-level analysis*.

Finally, the WCET for the entire program is calculated by finding the program path that takes the longest time, given the constraints of the flow information and the execution time of each basic block from the low-level analysis.

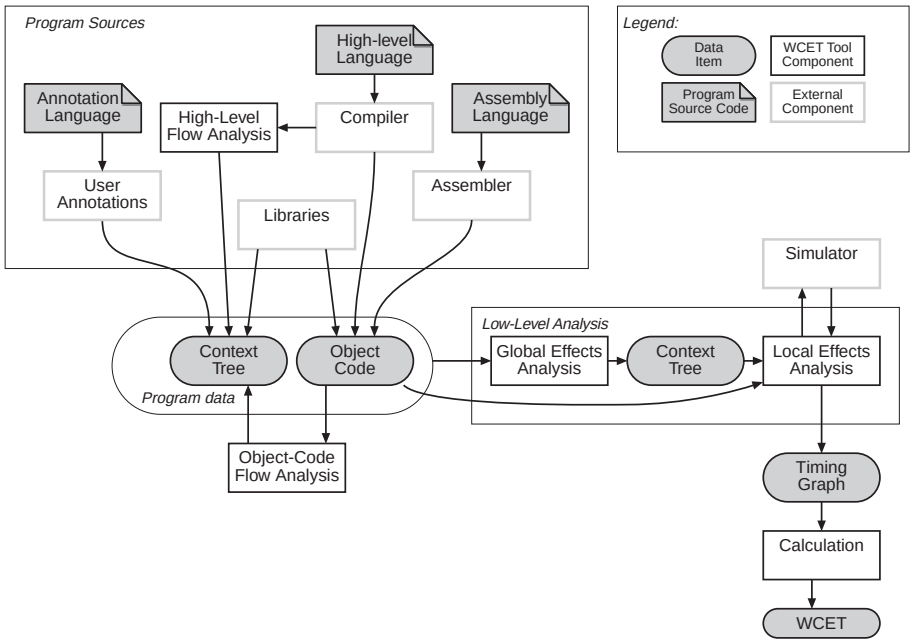


Figure 2: Our architecture for a WCET analysis tool

3 Architectural Overview

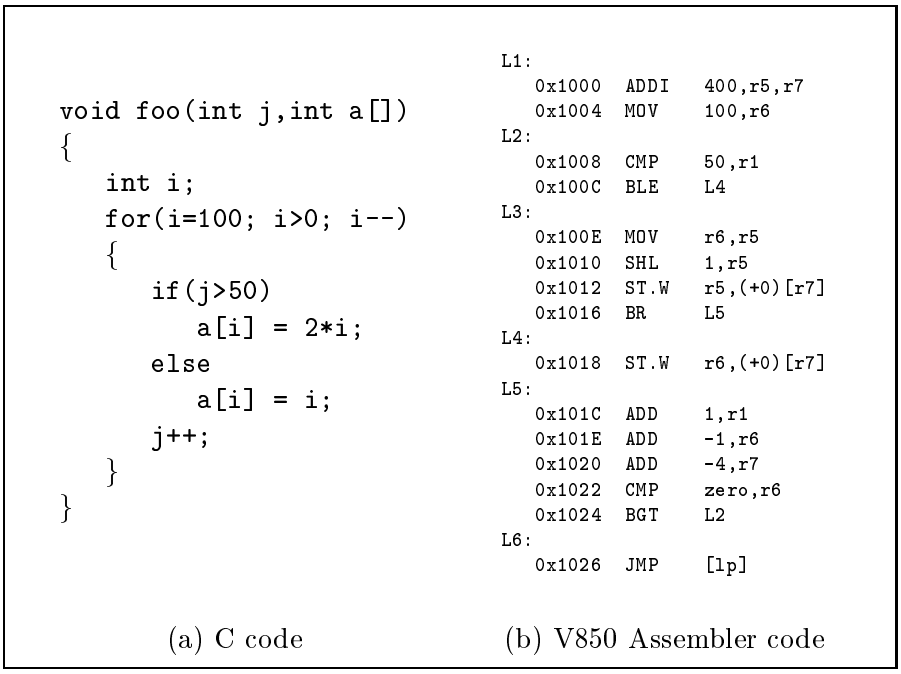
This section presents an overview of our proposed architecture for WCET analysis. The architecture integrates a variety of tools and methods in order to accommodate many different development scenarios. An illustration of the architecture is shown in Figure 2.

The architecture allows program sources (input) to be given in a combination of high-level language code, assembler code, and annotations. From these inputs, object code and a *context tree* containing the program flow information is generated. Low-level analysis is then applied to generate a *timing graph* (containing execution times for basic blocks and executable program paths) from which the WCET can be calculated.

In the following subsections, we will introduce the components in the architecture. A more detailed discussion on methods and data structures is left to Sections 4 to 7.

We will use the the example program in Figure 3 to illustrate the process of WCET analysis in our architecture. Note that the chip we have used in the example, the NEC V850 [NEC95], has both 16- and 32-bit instructions. Also note that the loop has been compiled into a `do-while`-loop, since it always iterates at least once. For the flow analysis, we assume that the parameter `j` is limited to the range [1..10] (this information could be provided by the user, or by automatic program analysis).

[1] A basic block is a piece of code that is executed in sequence (contains no jumps or branches, and there are no jumps into the sequence).



```
void foo(int j,int a[])
{
   int i;
   for(i=100; i>0; i--)
   {
      if(j>50)
         a[i] = 2*i;
      else
         a[i] = i;
      j++;
   }
}
```

(a) C code

```
L1:
   0x1000  ADDI   400,r5,r7
   0x1004  MOV    100,r6
L2:
   0x1008  CMP    50,r1
   0x100C  BLE    L4
L3:
   0x100E  MOV    r6,r5
   0x1010  SHL    1,r5
   0x1012  ST.W   r5,(+0)[r7]
   0x1016  BR     L5
L4:
   0x1018  ST.W   r6,(+0)[r7]
L5:
   0x101C  ADD    1,r1
   0x101E  ADD    -1,r6
   0x1020  ADD    -4,r7
   0x1022  CMP    zero,r6
   0x1024  BGT    L2
L6:
   0x1026  JMP    [lp]
```

(b) V850 Assembler code

Figure 3: C-code with corresponding assembler code

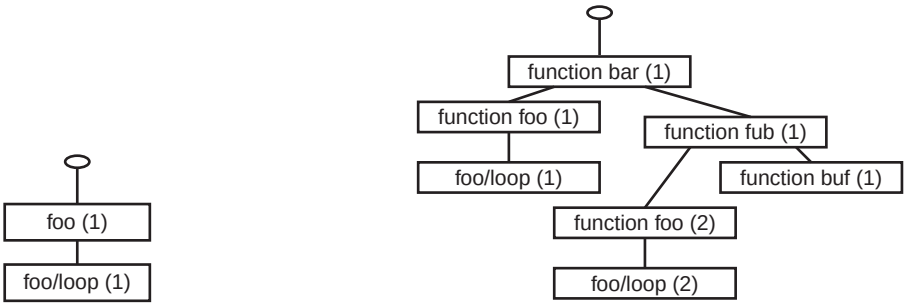
3.1 Program Sources

The program sources are anything that might generate object code and/or flow information. In Figure 2 we have included the most common sources.

- A *compiler* converts high-level language code into object code. To make it possible to take advantage of the program analysis performed by the compiler, we include a *flow analysis* module which collects information from the compiler and performs WCET-specific program analysis based on the collected information. The flow analysis module should be tightly coupled to the compiler, in order to facilitate tracing of the effects of compiler optimizations and to take advantage of the compiler's program analysis.

- An *assembler* converts assembler source code into object code. In our view, an assembler is a rather dumb system that cannot provide flow information for the code. Instead, manual annotations or object code flow analysis is used to provide flow information for the assembler code.

- *Libraries* are provided by compiler vendors, operating system vendors, and other third party providers. In order to make use of libraries in WCET-analyzed programs, we must have information of the program flow in the libraries. Flow information is either specified by the library provider, or deduced from the object code using object-code flow analysis.



(a) Context tree for the example from Figure 3. (b) A more complex context tree.

Figure 4: Context trees

- *User annotations* is a simple way to get information about the flow of a program. The programmer manually provides information about the program flow.

We aim to accommodate all ways in which program code and flow information may enter a project. Most industrial programming projects include a mix of hand-written assembler code, high-level language code, and libraries. Hence, an industrial WCET tool must support the integration of code from several different sources in the same program.

3.2 Object-Code Flow Analysis

One way to obtain program flow information is to perform analysis on the object code. Performing analysis on the object code level makes sense for programs containing a high proportion of assembler code.

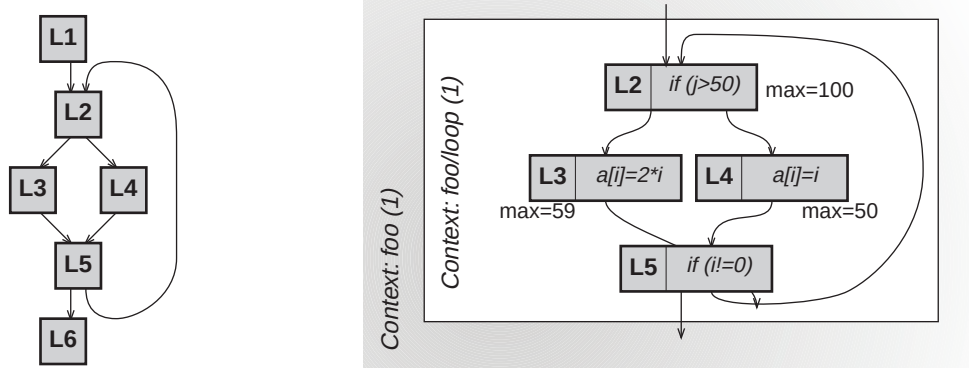
For programs compiled from high-level code, object code analysis is likely to give a worse result than analysis in conjunction with the compilation process, since most of the information inherent in the source code is thrown away when object code is generated. It is (much) harder to understand the intents of the programmer from the compiled and optimized object code. However, object-code flow analysis can be used in cases where it is impossible to modify the compiler to access its internal data.

3.3 Program Data

All information about the program to be analyzed is gathered in the *Program Data* data structure, see Figure 2 on page 4. The program data consists of two parts, the object code and the context tree (containing flow information).

Following the ideas by [FMW97, HAM+99], we view a program as a tree of function calls and loops (and maybe other types of flows). We call each node in this tree a *context*. The tree is called a *context tree*.

The contexts allow us to distinguish between different environments in which a certain basic block can be executed. For example, two different calls to the same function may



(a) The flow graph for function `foo` (b) Context and analysis results for `foo/loop(1)`

Figure 5: Flowgraph and contexts for the example function.

have very different executions (and thus very different execution times) depending on the values of the function arguments. Loops are often considered as contexts.

Each context contains a list of the basic blocks belonging to the context, and the flows between them (as illustrated in Figure 5(b)). Furthermore, information about limits to the flow, like loop bounds and mutually exclusive paths can be added (e.g. based on information from the flow analysis).

The context tree for the example program in Figure 3 is shown in Figure 4(a). Note that the loop is a context of its own, and that the nodes are numbered to indicate the possible existence of other instances of the function and the loop, as illustrated in Figure 4(b). The flow graph (the structure of the object code) for our example program is shown in Figure 5(a). Figure 5(b) shows the content of the loop context (the node `foo/loop(1)` in the tree in Figure 4(a)). The context includes information from the program analysis: the loop iterates at most 100 times (the information about block `L2`), the branch `L3` is taken at most 59 times, and `L4` at most 50 times (this is a consequence of limiting the value of `j` to [1..10]).

3.4 Low-Level Analysis

As described in Section 2, the purpose of low-level analysis is to determine the execution time for each basic block in the program. We partition low-level analysis into global and local effects analysis, presented below.

3.4.1 Global Effects Analysis

The global effects analysis considers the execution time effects of machine features that may reach across the *entire program*. Examples of such factors are instruction caches, data caches, branch predictors, and translation lookaside buffers (TLBs) [HP96].

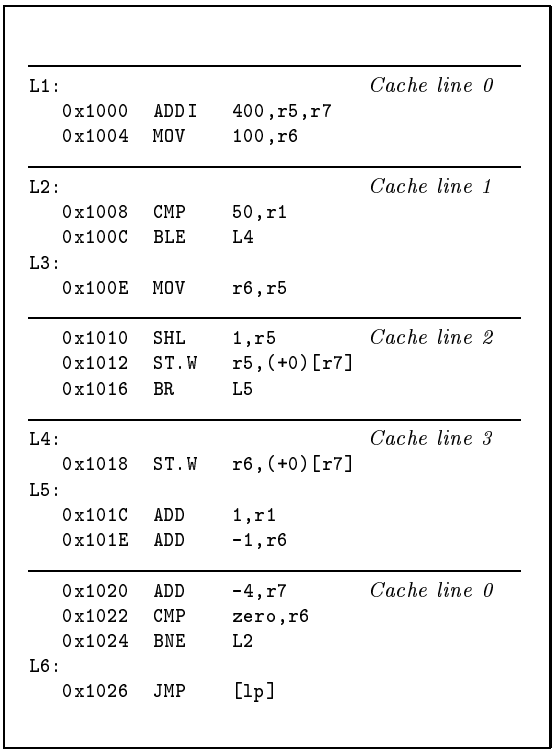


Figure 6: Cache layout of example program.

The global effects analysis uses the object code of the program and the flow information in the context tree to extend the context tree with the execution time effects of the considered machine features. It is sometimes necessary to add new contexts to express certain timing effects.

As an illustration, consider the example program in Figure 3 and a target CPU having an instruction cache with a cache line size of eight bytes, and four cache lines. The cache layout for the example program is shown in Figure 6.

The result of cache analysis for this example in the style of [FMW97] is shown in Figure 7. Each instruction has been categorized as either a cache hit or a cache miss. The cache analysis splits the loop into two contexts, "first iteration" and "successive iterations". We have highlighted four hits in the "successive iterations" that were misses in the "first iterations".

3.4.2 Local Effects Analysis

The local effects analysis handles machine timing effects that depend on a single instruction and its immediate neighbors. Examples of such effects are pipeline overlap (see for example [LBJ+95]) and memory access speed.

The local effects analysis takes the context tree from the global effects analysis and the object code for the program, and generates the timing graph. In the timing graph, each

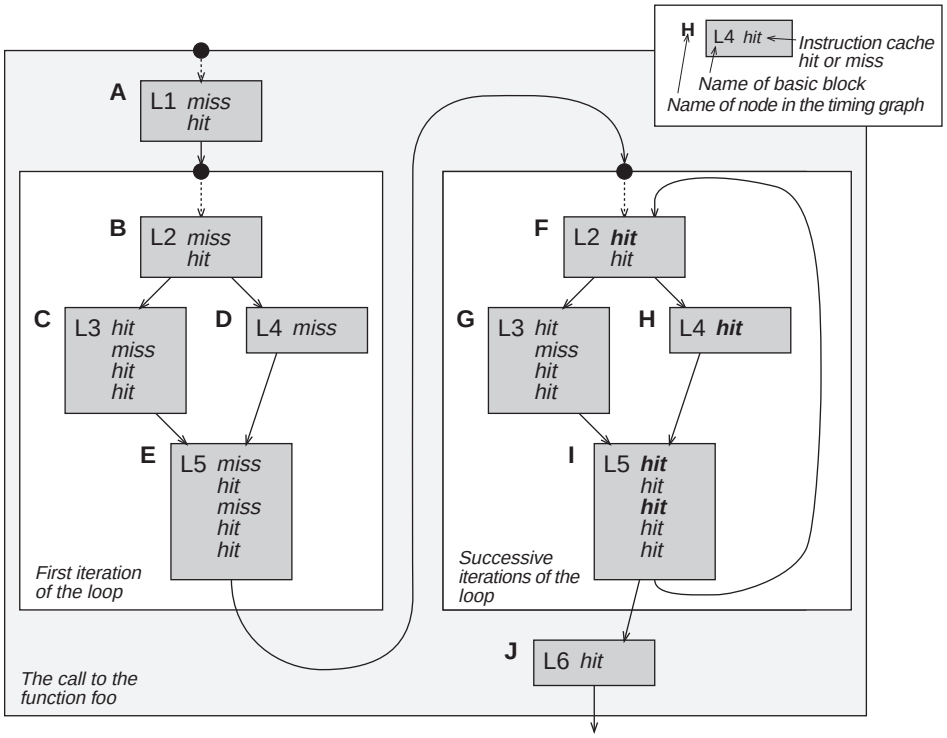


Figure 7: The results of the cache analysis for the example program.

basic block is present in a copy for each context in which it occurs, since it may have a different execution time for each context.

The calculation of concrete execution times is performed by sending the code to be executed to a simulator for the target architecture. For each basic block, the simulator is provided with an *execution scenario*. The execution scenario provides information about the execution of each instruction in the basic block, like whether the instruction hits or misses the instruction cache (as generated by the global effects analysis), the speed of the memory accessed, and operand values (from the flow analysis). Whenever a piece of information is missing, worst-case assumptions are used.

3.5 The Simulator

The simulator used in the local effects analysis could be tailor-made for WCET analysis, but preferably, it is a standard simulator provided by the hardware vendor, as part of the compiler, or by a third party simulator specialist.

The simulator should be trace-driven and cycle-accurate, but it does not need to emulate the semantics of the instructions, since program semantics is handled by the flow analysis. For the low-level analysis, only execution times are of interest.

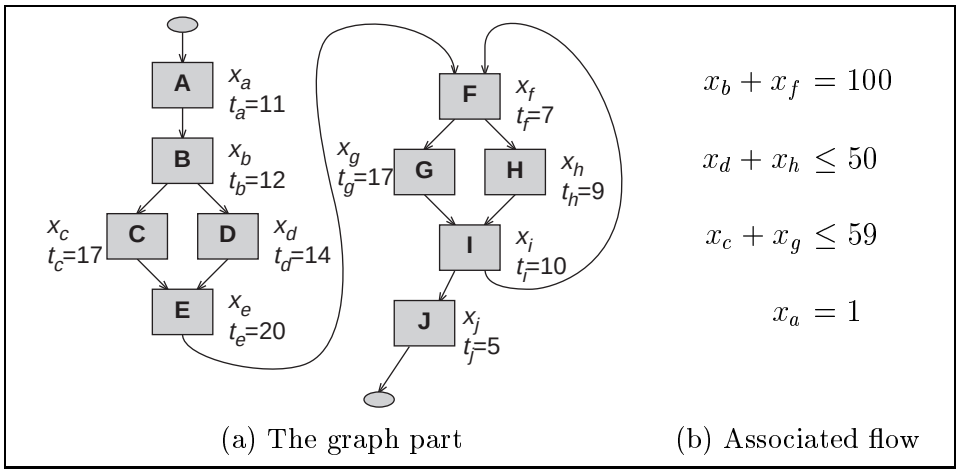


Figure 8: The timing graph for the example program.

3.6 The Timing Graph

The timing graph combines the program flow with the basic block execution times. Each node in the timing graph corresponds to an execution scenario and has an associated execution time (t_{block}). The execution scenario is provided by the context tree – one execution scenario is generated for a basic block for every context in which it occurs.

For pipelined CPUs, the execution time is the time to execute the basic block in isolation. The overlap with other basic blocks is represented by gain terms on the edges connecting the two blocks (g_{edge}), as described in [OS97].

The program flow information is expressed by constraints on the execution count variables (x_{block}) for edges and basic blocks.

The timing graph for our example program is shown in Figure 8. Figure 8(a) shows the graph with nodes, edges, and execution times (for simplicity, we have excluded the gain terms). Figure 8(b) shows the constraints on the flow. Note that the loop iteration count from Figure 5 (max 100 iterations) has been translated into the constraint $x_b + x_f = 100$ (the total number of executions of the basic blocks `B` and `F` is equal to 100), since the loop has been split into two contexts in the cache analysis.

3.7 Calculation

The calculator determines the final execution time for a program by solving the optimization problem obtained by maximizing a goal function derived from the timing graph. The goal function is the sum of all execution count variables multiplied with their respective execution times[2]:

[2]the gains are *subtracted* since the imply a speedup

Structural	Flow	Times	Gains
$x_a = x_{ab}$	$x_b + x_f = 100$	$t_a = 11$	$g_{ab} = 4$
$x_b = x_{ab} = x_{bc} + x_{bd}$	$x_d + x_h \leq 50$	$t_b = 12$	$g_{bc} = 4$
$x_c = x_{bc} = x_{ce}$	$x_c + x_g \leq 59$	$t_c = 17$	$g_{bd} = 3$
$x_d = x_{bd} = x_{de}$	$x_a = 1$	$t_d = 14$	$g_{ce} = 3$
$x_e = x_{ce} + x_{de} = x_{ef}$		$t_e = 20$	$g_{de} = 8$
$x_f = x_{ef} + x_{if} = x_{fg} + x_{fh}$		$t_f = 7$	$g_{ef} = 3$
$x_g = x_{fg} = x_{gi}$		$t_g = 17$	$g_{if} = 3$
$x_h = x_{fh} = x_{hi}$		$t_h = 9$	$g_{fg} = 4$
$x_i = x_{gi} + x_{hi} = x_{ij} + x_{if}$		$t_i = 10$	$g_{fh} = 3$
$x_j = x_{ij}$		$t_j = 5$	$g_{gi} = 4$
			$g_{hi} = 4$
			$g_{ij} = 4$

Figure 9: Constraint set for example program.

$$\sum_{\forall block} x_{block} \cdot t_{block} - \sum_{\forall edge} x_{edge} \cdot g_{edge}$$

This is known as the *Implicit Path Enumeration Technique* (IPET) [PS95].

The constraint set for our example program is shown in Figure 9. We have divided them into the following categories (from left to right in the figure): structural constraints (derived from the structure of the code), flow constraints (expressing limits to the program flow), execution times for basic blocks, and gains for edges. The structural constraints are used to represent the structure of the program flow graph (see e.g. [PS95]). In short, the number of paths entering a basic block is equal to the number of paths exiting the basic block.

Note that the basic block times and edge gains is based on the following assumptions: the CPU uses the V850E pipeline [NEC95], cache misses delay instruction processing by five cycles, and data memory accesses take five cycles in the `MEM` stage and do not interfere with instruction accesses[3].

4 Flow Analysis

The task of the flow analysis is to identify the possible ways a program can execute. The flow analysis gives us information about the program flow which is used in later stages of the WCET analysis. To facilitate a tight WCET estimate the flow analysis should

[3]The V850 chip does not include a cache. We have assumed one for illustration purposes.

be *context sensitive*, i.e., it should take advantage of the fact that variable values and parameters often have known values in a given context.

The flow information includes information about the maximum looping in the program (which is mandatory, since the WCET would otherwise be infinite), and information about *infeasible paths* (e.g. paths that cannot be executed because of data dependencies). Information about infeasible paths is used to make the analysis tighter, and is desirable but not mandatory.

Flow analysis can be carried out both on the object code and on the source code. In this section, we concentrate on the later approach, since better results can be obtained by analyzing the source code. This is often called *high-level analysis* in the literature.

In many WCET analysis methods [CBW94, LMW96, PS93, TF98], the flow information is given by *manual annotations*, i.e. extra information about the program flow and/or data dependencies is provided by the programmer.

We claim that manual annotations are error prone and should be avoided. It is easy to find examples where, e.g., the maximum number of iterations in a loop is difficult to calculate, and where data dependencies are complicated. Furthermore, programmers may write down what they *think* the code does, not what it *actually* does.

In recent years, there has been some research performed to automatically deduce program flow information, with the goal of reducing the dependence on manual annotations [Alt96, Cha95, EG97, HSRW98, HW99]. However, all of these methods are limited in the types of programs they can handle, and manual annotations probably have to be used in some cases. For example, the analysis method employed may not calculate the kind of information needed, or the automatically deduced information is not tight enough. As anybody who has used a compiler knows, there are cases when programming tools just cannot figure out what is blindingly obvious to the programmer.

4.1 Context-Sensitive Calculations

The simplest method for WCET analysis is to calculate just one WCET for a program or a function, disregarding the fact that the execution time can depend on the input parameters. This can lead to large overestimations of the WCET, making the result of WCET analysis less useful.

Instead, we would like the calculations to be *context sensitive*, i.e., they should take advantage of the fact that input parameters often have known limits, e.g. there may be physical limits to the possible values for inputs from the environment or a function may be determined by the context of the call.

If input data varies between different calls to a function, and if the WCET is not independent of the input values, the WCET should be calculated separately for each function call to yield a tighter WCET estimate. An example of such a function, with a heavy input dependence, is shown in Figure 12 on page 19.

The limits of input values to the program (values read from hardware sensors, etc.) have to be given by manual annotations. However, limits on the values of function parameters and variables inside a context should ideally be calculated automatically.

An alternative to handle of input sensitivity is to use *modes*, as presented in [CBW94]. A mode is a constraint specifying sets of input values to a function that cause identical program flow, which also means identical timing characteristics. However, the modes are not deduced automatically.

Note that in many cases, functions do not exhibit input-dependent timing behavior. According to [Eng99], about one third of the functions in a large selection of embedded real-time programs contained no decisions and would thus have constant execution time (unless individual instructions can have variable execution time). Such functions could be analyzed only once, to speed up the analysis process.

4.2 The Maximum Number of Iterations

The flow analysis should calculate a safe and tight approximation of the maximum number of iterations in loops. Knowing an upper bound for the number of loop iterations for each loop is required to obtain a finite WCET value.

For nested loops, the number of iterations of the inner loop should be estimated for *each* iteration of the outer loop. This is important to get a tight WCET. However, it might be hard to represent this information in an efficient manner.

The complexity of the loops will determine the difficulty of the loop bound analysis. In the most general case the automatic detection of the number of loop iterations is undecidable. In cases where we cannot determine the maximum number of iterations automatically we have to resort to asking the programmer for guidance (i.e. to provide manual annotations).

When recursion is used in a program, the maximum recursion depth need to be known (just as for loops). Recursion is usually avoided when programming real-time applications, due to potential memory problems and the belief that recursion is harder to understand and analyze than iteration [Jon99]. However, [Eng99] shows that recursive programs do occur in embedded real-time programs (even if it is quite uncommon). Our conclusion is that it is relevant to analyze recursion, but that it is not the most urgent problem to solve.

4.3 Information About Infeasible Paths

Infeasible paths are program paths that cannot be executed. A common form of infeasible path analysis, employed in compilers, is dead code elimination. In other cases, a piece of code may not be dead, but certain execution paths involving it may be impossible (given information about variable values and data dependencies). For example, two `if`-statements can be mutually exclusive. If infeasible paths can be excluded from the WCET analysis, the WCET estimate may become tighter.

An automatic analysis to find infeasible paths need not be able find each and every infeasible path. It is safe to miss some infeasible paths, but feasible paths must not be pointed out as infeasible, since this might lead to an underestimation of the WCET.

Method	Context sensitive calculations	Number of iterations in loops	Maximum depth of recursion	Infeasible paths
Uppsala	√	√	√[1]	√
Paderborn				√
York	√[2]	√[2]		√[2]
Florida	√	√[3]		√

[1] The function is supported by the method, but is not implemented.
[2] Some manual annotations are needed.
[3] The method handles special types of *for*-loops.

The methods:

Uppsala: Uppsala University and Mälardalen University, Uppsala and Västerås, Sweden. Program analysis (C, Smalltalk/RTT) based on abstract interpretation [EG97, GE98].

Paderborn: C-LAB, Paderborn, Germany. Program analysis (C) based on symbolic execution [Alt96].

York: University of York, York, Great Britain. Program analysis (Ada) based on symbolic execution [Cha94, Cha95].

Florida: Florida State University, Tallahassee, USA. Analysis of certain types of loops in C [HSRW98, HW99].

Table 1: Overview of flow information calculation capabilities

4.4 Automatic Flow Analysis

Program flow analysis is used extensively in optimizing compilers [Muc97]. However, the analysis performed in a compiler typically deduces other information than that needed for WCET analysis, and as a result, there has been some research into WCET-specific program flow analysis. Table 1 gives an overview of the capabilities of some of the flow analysis developed for WCET analysis.

We will use the example in Figure 3 on page 5 to demonstrate how the Uppsala method in Table 1 works (abstract interpretation over integer intervals). With this method we calculate the values of variables, the maximum number of iterations of loops, and infeasible paths in the program. The results are shown in Figure 10.

The flow analysis calculates these results by performing the following steps:

1. All variables that do not influence the control flow are identified and ignored in the analysis (e.g. as by [HW99]). In this case `i` and `j` determine program flow but the array `a` does not influence the flow.

2. Input values for the considered variables are accounted for (in this case we assume that $j = [1..10]$).

3. The values of the variables are approximated for each program point. This analysis uses abstract interpretation where integers are represented by intervals. The most important values are shown in Figure 10.

```
void foo (int j, int a[])          j = [1..10]
{
  int i;                           i = [−∞..∞]
  for (i=100; i>0; i--)            i = [1..100], x_L2 = 100
    {
    if(j>50)
      a[i] = 2*i;                  j = [51..109], 50 ≤ x_L3 ≤ 59
    else
      a[i] = i;                    j = [1..50], 41 ≤ x_L4 ≤ 50
    j++;
    }
}                                  i = [0..0], j = [101..110]
```

Figure 10: Example program with analysis results

4. The minimum and maximum number of iterations in the loop are calculated, and the corresponding constraint ($x_{\texttt{L2}} = 100$) is generated.

5. The infeasible paths in the program are calculated. In the example program, it is structurally conceivable that either `L3` or `L4` are executed on all iterations of the loop. However, taking the information about the `j` variable into account allows us to deduce that `L3` can be executed at most 59 times. The corresponding limit for `L4` is 50 times.

The information generated about the program flow is included in the final timing analysis (see Figure 9 on page 11). Note that the basic blocks on which the analysis was performed have been split into several timing graph nodes. E.g., $x_{\texttt{L2}}$ in Figure 5 on page 7 has been transformed to $x_b + x_f$ in Figure 8 on page 10.

4.5 Compiler Integration

As mentioned in Section 3.1, the compiler can provide important information for the flow analysis tool. Integrating the flow analysis with the compiler has several advantages:

- The compiler source-language parser can be used, which makes the flow analysis tool easier to write. This is the most basic form of compiler-analysis integration.
- The analysis can take advantage of the compiler's information about the program.
- It becomes easier to handle the problems introduced by optimizing compilers.
- The analysis can be performed on the intermediate code used in the compiler.

4.5.1 Using Program Information

All compilers perform extensive program analysis, in order to generate code and optimize the programs. The generated information is certainly useful for flow analysis, and making it available to the flow analysis tool will substantially facilitate the design of this tool.

4.5.2 Handling Optimizations

An important problem for flow analysis is how to handle the optimizations performed by a modern compiler. For a simple compiler, the source code and the object code have the same structure, which makes it easy to map the analysis results from the source code to the object code (this mapping is needed since the flow on the object-code level is used to calculate the WCET). For an optimizing compiler, the relation is non-trivial: flows can be changed, introduced, and deleted, and loops can be unrolled, inverted, split, etc.

To overcome this problem, the flow analysis must know how the program has been optimized. This requires cooperation between the compiler and the flow analysis tool. Some research has been performed on this [EAE98, LKM98].

4.5.3 Intermediate-Code Analysis

Flow analysis is usually considered to be performed on the source code of a program (C, Ada, or other high-level languages). This approach has the advantage of making it easier to understand the programmer's intentions, and facilitating communication with the programmer about the program. The major disadvantage of performing flow analysis on the program source code is that it can be quite difficult to map the information from the source code to the object code, especially in the presence of optimizing compilers (as discussed above).

The most extreme alternative is to perform the flow analysis on the object code, which makes the mapping problem disappear. However, the object code contains much less information than the source code, which makes an object-code analysis potentially less exact.

A novel solution to the problem is to perform the flow analysis on the compiler's intermediate representation of a high-level program. This has several advantages:

- The intermediate language is typically simpler than the source language. Error checking, preprocessing, and other complex source language processing has already been performed.
- The intermediate code contains all the information from the source code. No information is lost, as is the case with the object code.
- The intermediate code has the same structure as the final object code (provided that we perform the analysis after optimizations have been performed).

- Program constructs that may cause complex processing invisible on the source code level (e.g. array assignments, where loops or library function calls may be generated) are made explicit, allowing them to be analyzed together with the rest of the program.
- Many compiler products use the same intermediate language for several source languages (for example Digital's GEM compilers [BCD+92]), and thus the flow analysis would not be tied to a certain high-level language.

5 The Context Tree

The *context tree* is the central depository for program flow information, and reflects the dynamic execution characteristics of the program. The definition of what constitutes a context depends on the analysis methods employed. Our architecture does not specify which definition to use.

For example, in the WCET analysis presented by [HAM+99], the contexts are loops and functions. In the cache analysis presented by [FMW97], loops and recursive functions are split into their first and other iterations, leading to four different types of contexts (loop first, loop other, function call, and recursive function call). Alternatively, the context tree could store information on the form of a timing schema [LBJ+95, PS90, PK89], where the nodes correspond to loops and conditional statements.

Given a reasonable definition of contexts, the context tree is much smaller than a complete unrolling of the execution of the program. Using one node (a loop context) to represent all iterations of a loop, for example, is much more compact than representing each iteration separately.

All flow analysis methods used in the WCET analysis must agree on the definition of a context. An analysis method may use some other internal program representation, but it should store its results in a context tree using the same context definition as the other WCET analysis modules. Note that the context tree only contains info useful to the succeeding analyses.

Regardless of the definition of a context, our architecture specifies that each context must contain the following information:

- An entry node.
- A finiteness limit.
- The code belonging to the context.

In addition, a context can contain optional information to make the WCET analysis tighter.

The context tree as presented here is powerful and flexible enough to handle the flow information generated by the leading WCET analysis methods.

5.1 Flow Information

There are two kinds of flow information: finiteness limits (for example loop bounds), and optional information intended to tighten the analysis (for example infeasible paths).

The finiteness limit is expressed as a limit on the number of times the entry node can be executed for each time the context is entered. The purpose of this limit is to guarantee that the context tree represents a finite execution of the program. This means that a loop context must have the loop header (the first block in the loop) as its entry node.

The optional flow information (such as infeasible paths) is intended to limit the set of possible executions, making the WCET analysis tighter. An example of optional flow information is the bounds on the number of executions of blocks `L3` and `L4` in Figure 5 on page 7.

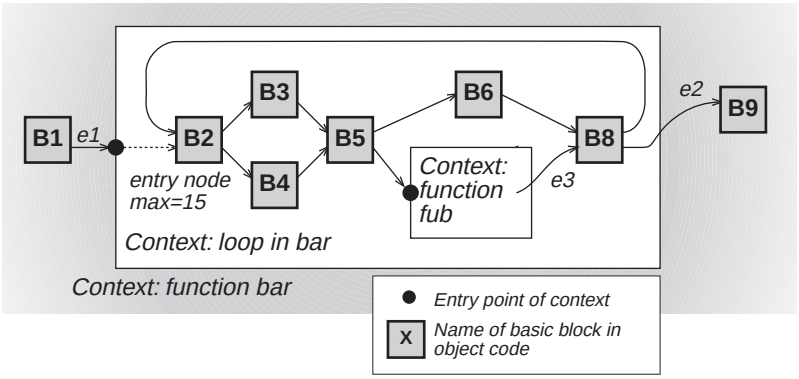


Figure 11: Example loop context

Figure 11 shows an example context, with fragments of the surrounding context, and a subordinate context (whose content is not shown). The entry node is `B2`, and it has a finiteness limit (`max=15`).

5.2 Information for Low-Level Analysis

For the purpose of low-level analysis, each context contains the basic blocks included in the context, and their connecting edges[4]. Some edges may exit the context, leading either to specified basic blocks in surrounding contexts, or to the entry point of some sub-context (for example, edge `e1` in Figure 11 points to the entry for the loop context, and edges `e2` and `e3` exit contexts).

It is also possible to specify information contributing to the execution scenarios of the basic blocks. For example, for a CPU where instruction execution times varies depending on operand values, limits on possible operand values could be entered into the context.

[4]This is most efficiently implemented by references to the object code.

```
short doop(enum OP op, short a, short b)
{
   switch(op) {
      case PLUS:
         return a + b;
      case MINUS:
         return a - b;
      case TIMES:
         return a * b;
      ...  etc.
   }
}
```

Figure 12: Example function with contextually dead code.

Each context corresponds to a certain concrete execution of a part of the program, and contains the basic blocks executed in that part of the program. Figure 5 on page 7 shows how a loop context contains copies of the blocks in the loop.

The set of basic blocks in a context need not be the complete set of basic blocks in the flow graph for the function or loop. Consider the function in Figure 12, where the `op` parameter is used to select one action from many possible. If the value of `op` was known to be `PLUS` for a particular function call, we would only need to include the code for the `PLUS` operation in the context. This is an example of context-sensitive flow analysis.

5.3 Unstructured Flow Graphs

Note that in order to handle some programs, we will need to have contexts for *unstructured flow graph fragments* (see [Muc97], page 196).

As shown by [Eng99], unstructured flow graphs do occur in real embedded real-time programs[5]. No WCET analysis method in the literature allows unstructured flow graphs.

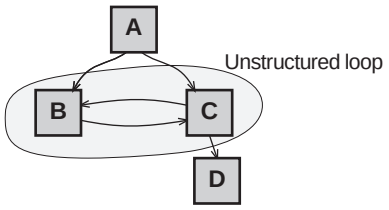


Figure 13: A simple unstructured loop

The main problem is with unstructured loops. As illustrated in Figure 13, such loops

[5]The unstructured flow graphs were both written by the programmer and generated by compiler optimizations.

have no well-defined header node, since they can be entered at several places. We plan to handle such unstructured loops by duplicating the loop: we use one context containing the code for the unstructured loop for each entry to the loop, and thus we obtain the single entry node needed to create a context.

6 Low-Level Analysis

The purpose of low-level analysis is to account for hardware effects on the execution time. The clear separation of program flow analysis from the hardware dependent low-level analysis allows each analysis to be specialized for its target problem domain. For example, the flow analysis can be performed without considering the hardware timing characteristics.

6.1 Global Effects Analysis

The global effects analysis handles machine features that must be modeled over the whole program to be correctly predicted. The global analysis determines how global effects affects the execution time, but it does not generate concrete execution times.

6.1.1 Separation of Global and Local Analysis

Our architecture assumes that all global effects are determined before we enter the local effects analysis. Thus, we assume that the results of global analysis do not depend on, for example, the exact state of the pipeline for a certain instruction. In addition, we require a well-defined worst case for each global effect, e.g., that a cache miss is always worse than a cache hit, etc. In case these assumptions do not hold, we can resort to modeling a "guaranteed" worst case, where we use a slight overestimation to produce a safe WCET estimate.

Some low-level analysis approaches have integrated the analysis of local and global effects [LBJ+95, OS97], while our approach is more reminiscent of [HAM+99, TF98], where the analysis of caches is performed as a separate stage.

6.1.2 Instruction Cache Analysis

Instruction cache analysis is the most well known example of global effects analysis, e.g., described in [HAM+99]. Their approach is to categorize each instruction cache access into one of the following categories: *first-miss*, *always-miss*, *first-hit* and *always-hit*. The analysis distinguishes between the first and successive executions of an instruction within a loop. A *first-miss* categorization means that the first execution of an instruction will generate an instruction cache miss, while in all successive executions the instruction will be in the cache. An *always-hit* categorization means that the instruction always will be in the cache.

Our categorization of the instruction in the `L4` block in the context graph in Figure 7 corresponds to the *first-miss* categorization and the first instruction in the `L3` block corresponds to an *always-hit*.

Other research groups have extended instruction cache analysis to include associative caches [WMH+97, Mue97a, FMW97], nested loops [Mue97b, TF98], and multi-level caches [Mue97b].

Note the categorization of the second instruction in the block `L3` given in Figure 7 on page 9. The classification is made accordingly to the methods outlined in [HAM+99, TF98] which are *iteration*-based loop analyses. I.e., since we can't know if the instruction has been referenced during its first iteration in the loop we must safely classify it as an instruction cache miss in both the first- and successive loop contexts. We would really like to classify the instruction as a *first-reference-miss*. Thus, after the first *reference* of the instruction we know that it always will be in the cache. We therefore conclude that the iteration-based instruction categorization is sometimes pessimistic and a tighter timing estimate can be achieved if we do a *reference*-based instruction cache analysis.

6.1.3 Other Global Effects Analyses

In addition to instruction cache analysis other hardware features that can be analyzed include data caches, unified caches, translation lookaside buffers, dynamic branch predictions and dynamic speculative executions [HP96].

Data cache analysis has been proposed in [WMH+97, Whi97][6]. The results can typically be formulated as "every i:th execution of an instruction will generate a data cache miss". As an illustration consider the example program in Figure 3 on page 5. Assuming that we have an 8 word data-cache line and no conflict between `A[i]` and the other data-items (`i` and `j`), a data cache analysis should report that every 8:th *reference* to `A[i]` will generate a data-cache miss. This is expressed in the context graph by annotating the instruction referencing `A[i]` as a "miss once, hit seven" data cache reference.

6.1.4 Changing the Context Tree in Global Effects Analysis

The global effects analysis may *change* the context tree from the program data. The most common change will be to add contexts in order to express the execution time effects of certain global factors.

The basic idea behind the context tree is to distinguish between the various contexts in which a basic block can be executed. Two different contexts for the same basic block are supposed to reflect some difference in the execution of the basic block. Such differences can occur both due to program flow (function calls etc.) and low-level hardware effects (caches etc.). The contexts relevant for low-level effects analysis may be very different from those relevant for flow analysis, and thus the global effects analysis may need to introduce new contexts.

[6] Others, like [LMW96, OS97] have also performed data cache analysis, but not as a separate analysis activity.

For example, in instruction cache analysis, it is natural to distinguish between the first and other iterations of a loop, since the code in the loop is likely to be loaded into the cache during the first iteration and then stay there during the other iterations. For other effects, other distinctions may be relevant. For example, for branch prediction analysis, the last iteration of a loop is likely to behave differently, since the branch prediction is likely to miss. In theory, a very exact loop analysis could be performed by making every potential iteration of a loop into a context of its own (which is prohibitively expensive in terms of storage).

6.2 Local Effects Analysis

The local effects analysis analyzes the instruction timing factors that can be analyzed locally (using only the neighboring instructions) and generates the timing graph.

6.2.1 Generating the Timing Graph

The input to the local effects analysis is the context tree from the global effects analysis and the object code of the program to analyze. The local effects analysis generates the timing graph from this information by traversing the complete context tree, generating both execution times for basic blocks and the timing graph. The hierarchy of the context tree is flatted into a graph.

6.2.2 Execution Scenarios

The central concept in our local effects analysis is the *execution scenario*. An execution scenario is a concrete execution of a basic block, with all unknown factors set to the worst case for the context of the basic block. Examples of potentially unknown factors are cache hits or misses, branch prediction results, whether branches are taken or not, the memory addresses or memory areas accessed by loads and stores, and the values of operands.

Each basic block execution scenario corresponds to a node in the timing graph, and generates one unique execution time.

6.2.3 Node Splitting inside Contexts

In some cases, one basic block may have several execution scenarios in the same context, for example to express data cache classifications like "one miss, seven hits", where it is hard to define reasonable contexts. In this case, the timing graph nodes are *split* until a unique execution scenario is obtained. This split is performed by the local analysis, as a part of the process of generating the timing graph.

In Figure 14 we show a simple example: a *first-miss* instruction cache classification (for the first instruction in block `B`) can be handled without defining a separate context for the first and other iterations of the surrounding loop[7]. We split the `B` node into the two

[7] We assume that the block `B` resides inside a loop (otherwise the classification would be meaningless).

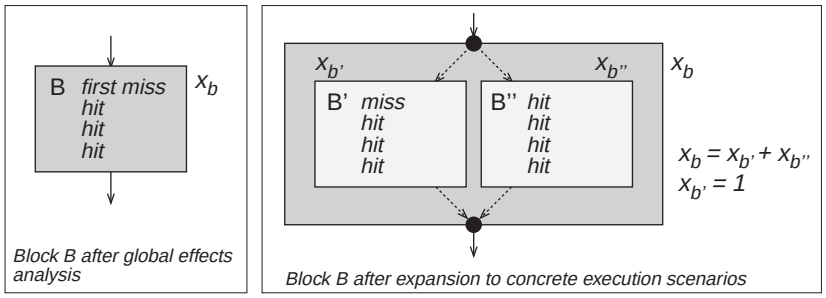


Figure 14: The expansion of a block to handle a "first miss" cache classification.

nodes `B`' and `B`". The sub-node `B`' represents the case that the first instruction misses the cache, and `B`" the case when it hits the cache. Without changing the surrounding timing graph, we add constraints describing how the executions of block `B` is divided between the sub-nodes.

The constraint $x_{b'} = 1$ represents that a *first-miss* will only generate a single cache miss. The fact that the two sub-nodes replace the `B` node in the execution of the program is expressed by the constraint $x_{b'} + x_{b''} = x_b$. All existing constraints on x_b can be kept unchanged.

The node-splitting concept is powerful enough to express the results from a reference based instruction cache analysis and the results from a data cache analysis.

6.2.4 Determining Execution Times by Simulation

We determine the time for each basic block execution scenario by executing it in a simulator. The result of the simulation run is an execution time, (t_{block}), which we store in the timing graph and use in the final execution time calculation.

6.2.5 Analyzing Pipeline Overlap by Simulation

One of the primary problems in low-level analysis for pipelined processors is to determine the overlap between two successive basic blocks. Traditionally, this has been performed by determining a pipeline state for both blocks, and then concatenating them (pipeline concatenation is for example presented by [LBJ+95, OS97]).

In our architecture the pipeline overlap is determined in a step separate from the final calculation of execution times, and the overlap is represented in the timing graph.

This is achieved by using the *gain modeling* method introduced in [OS97], which associates execution times not only with the nodes in the graph, but also with the *edges* connecting the nodes. The time on an edge is called a gain, and represents the amount of pipeline overlap between the two nodes (basic blocks) connected by the edge. In some cases, it might be necessary to consider timing effects across more than two blocks. Our implementation handles these cases, but the details are beyond the scope of this paper.

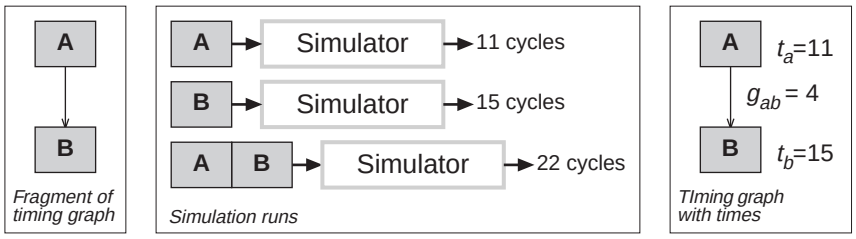


Figure 15: Analyzing pipeline overlap by simulation.

We obtain the overlap by executing the two basic blocks in sequence, and comparing the execution time to the total execution time of the two blocks in isolation, as illustrated in Figure 15. First the blocks `A` and `B` are executed in isolation, obtaining the times $t_a = 11$ and $t_b = 15$. Then the *sequence* $\langle$`A`,`B`$\rangle$ is executed, and the time $t_{ab} = 22$ is obtained. Since this value is smaller than the sum of t_a and t_b, we conclude that the execution of the blocks `A` and `B` overlaps. The amount of overlap is the *gain* when executing the edge `A`$\rightarrow$`B`, and is calculated as $g_{ab} = t_a + t_b - t_{ab}$. This number is entered into the timing graph and will be used in the final WCET calculation (see Figure 9 on page 11 for an illustration).

7 Calculation

The purpose of the *calculator* is to calculate the final WCET estimate for the program. A calculation method suitable for our WCET architecture should be:

- Expressive - it should be possible to include the results from many different types of analysis in the computation.
- Extendable - future analysis results and new hardware features should easily be integrated.
- Retargetable - it should be target-hardware and high-level language independent.
- Efficient - the calculation should terminate within a reasonable time.
- Safe and tight - the calculation should not add any pessimism of its own, and it should not cause the final timing estimate to become unsafe.

To speed the analysis it is advantageous if we can express and exploit the characteristics of the problem domain. Furthermore, we like to use a well established methodology for performing calculations, since this makes it easier to find good calculation modules.

When considering the analysis of hand-written assembly language and optimized high-level code, it is necessary that the calculation can handle unstructured flow graphs. The need for this is demonstrated by [Eng99].

7.1 Separation vs. Integration

A number of different calculation methods have been proposed in the literature. In some, the calculation of the final WCET is integrated with the low-level analysis. For example, both the global and local effects analyses are integrated with the WCET calculation in [LBJ+95, LMW96, OS97]. Another example is [LS98] where both the analysis of program flow and low-level effects analysis are integrated with the WCET calculation.

We believe that for a WCET tool to be able to integrate new analyses and methods, the WCET calculation should be separate from the analyses. Trying to integrate several different analyses in the calculation is likely to cause an explosion in complexity.

	Calc Method			Analyses Performed		
	Path	Tree	IPET	Flow	Global	Local
Florida	√			•	•	×
Gothenburg	√			×	×	×
Paderborn	√			×	•	•
York	√			•	–	–
Seoul		√		–	×	×
Vienna$_{old}$		√		–	–	–
Vienna$_{new}$			√	–	–	–
Princeton			√	–	×	×
Saarbrücken			√	–	•	–
Uppsala$_{old}$			√	–	×	×
Uppsala$_{new}$			√	•	•	•

- • Result of separate analysis used in WCET calculation
- × Analysis integrated in WCET calculation
- – No analysis performed

The research groups:

Florida:	Florida State University, Tallahassee, USA. [HAM+99, WMH+97, Mue97a]
Gothenburg:	Chalmers University of Technology, Gothenburg, Sweden. [LS98]
Paderborn:	C-LAB, Paderborn, Germany. [Alt96, SA97]
York:	University of York, York, Great Britain. [CBW94, Cha95]
Vienna:	Technical University of Vienna, Vienna, Austria. [Vrc94, PS95]
Seoul:	Seoul National University, Seoul, Korea. [LBJ+95, LKM98]
Princeton:	Princeton University, Princeton, USA. [LM95, LMW96]
Saarbrücken:	Universität des Saarlandes, Saarbrücken, Germany. [TF98, FMW97]
Uppsala:	Uppsala University, Uppsala, Sweden. [OS97, EAE98]

Table 2: Overview of calculation methods

7.2 Calculation Methods

We can (roughly) divide the calculation methods proposed in the literature into three categories: path-, tree-, or IPET[8]-based.

[8]Implicit Path Enumeration Technique

The path-based category includes calculation methods that generate the final WCET estimate by calculating times for different paths in a program, searching for the path with the longest execution time. The defining feature is that possible execution paths are *explicit* represented.

In tree-based methods the final WCET is generated by a bottom-up traversal of a tree representing the program. The analysis results for smaller parts of the program is used to build up timing estimates for larger parts of the program.

Methods in the IPET-based category express the dynamic program flow and the hardware timing effects using constraints. The final WCET estimate is calculated by maximizing a goal function that ties the constraints together. The maximization is usually performed using either constraint satisfaction methods (CSP) or integer linear programming (ILP). The difference from the path-based methods is that possible program paths are handled *implicitly*. Only the fact that a certain basic block is executed a certain number of times is expressed, not the exact paths causing that basic block to be executed.

We summarize the calculation methods used by various WCET research groups in Table 2. We have named the research groups according to the city (or state) where they are located.

We distinguish between the early work (Vienna$_{old}$) of the research group in Vienna where a tree-based calculation was used [Vrc94] and their later work (Vienna$_{new}$) where an IPET-based calculation method is used [PS95]. We also distinguish between our previous approach to WCET calculation (Uppsala$_{old}$) [OS97] and our current approach (Uppsala$_{new}$) as described in this paper.

7.3 The Implicit Path Enumeration Technique

We have chosen IPET as our calculation method since it is the method that best satisfies our requirements. We believe that path- and tree-based methods are harder to retarget and have more problems in integrating results from different program flow and low-level analyses compared to IPET-based methods.

For example, tree-based methods have problems in expressing flow constraints that reach over a larger part of the program, and cannot handle unstructured flow graphs. Path-based methods suffer problems for programs with large number of possible paths, since each path has to be explicitly considered. Furthermore, integrating the results of new analyses usually requires changing the implementation of the calculation module.

The IPET calculation is easy to extend to handle new analysis methods. The results of new analyses are expressed either as constraints on the flow (for flow analysis methods), details in execution scenarios (for global effects analyses), or execution times in the timing graph (for local effects analyses). The calculation method does not need to change at all, thanks to the careful separation of calculation and analysis, and the expressive power of the IPET formulation.

Another advantage of the IPET method is that it produces a profile of the worst-case execution of the program. The solution to the maximization problem is a set of concrete values for the execution count variables, which shows how many times various

basic blocks and edges are executed in the worst-case execution. This could be used to guide optimizations to reduce the worst-case execution.

The quality of the final WCET estimate depends on the quality of the analysis methods employed. We believe that if we include the same information in the IPET formulation as used by other calculation methods, the result should be of at least the same quality.

The possible drawback of the IPET method is that, since the problem has been abstracted, it is rather hard to develop algorithm heuristics that use knowledge of the problem domain.

There are two dominant solution methods used for IPET:

- Integer Linear Programming (ILP) as in [LM95, PS95, TF98].
- Constraint Satisfaction (CSP) [Tsa93] as in [OS97].

The benefits of using ILP is that it is very efficient (several good commercial solvers exist [CO99]) as long as linear constraints can be used to express the results of our analyses. The benefits of using CSP is that we can express more complex (non-linear) constraints and that we can specify search heuristics. By adding redundant constraints to the CSP problem we can also make the solution process more efficient.

8 Future Work

Our long term goal is to produce a usable WCET tool, available as shrink-wrapped software on the real-time market. To this end, we cooperate with IAR Systems (Uppsala, Sweden), a vendor of embedded system programming tools. We hope to be able to integrate WCET analysis into their integrated development environment. We believe that this integration with accepted tools is the best way to get WCET analysis accepted on the market, and to make practioners in the real-time field actually use execution-time analysis.

In our continued work, we plan to to address the issues described in the following sections.

8.1 Component-Based WCET Analysis

In the architecture sketch in Figure 2 on page 4, *libraries* are listed among the program sources. The main problem with libraries is that they are usually delivered as object code, thus prohibiting source-code flow analysis. To integrate the libraries into our WCET analysis framework, we need to provide context subtrees containing flow information for the library calls.

One way to handle this is by flow analysis on the object code of the library. However, since object-code analysis is expected to give less information than analysis on the source-code level, this is not a preferable approach.

An alternative approach would be to integrate analysis of library components in the compilation of the library. That is, we want to be able to analyze *program components*, and then use the results as components in the final analysis of the complete application.

This would have several benefits:

- A library vendor could provide information about their components to allow tight execution time estimates, without compromising their source code.
- Stable parts of the application could be converted into components, which would reduce the amount of work required for future WCET analyses of the application.

Finding a good model for *component-based WCET analysis* is one of our main future research goals.

8.2 Partial Program Analysis

Traditionally, WCET analysis is performed on the completed, compiled and linked program. However, the entire program may not be available for analysis. Engineers would like to be able to check small parts of a program before the program can be linked in its entirety. Maybe parts of the development is performed by sub-contractors, whose code is going to be linked in quite late. In those cases, support for *partial program WCET analysis* is needed.

Note that partial program analysis and component-based WCET analysis have much in common, but that they address two different problems: partial program analysis allows an incomplete program to be analyzed, while component-based analysis allows complete components to be efficiently reused. Given component-based WCET analysis, it should be quite easy to implement partial program analysis.

A problem with WCET analysis is that it is essentially about properties of complete programs or closed pieces of programs. A piece of code calling an unknown piece of code really cannot be timed.

A reasonable solution is to allow the user to specify timing information for the fragments of a program which are not available. Such abstractions should be specified on the level of functions (and include information about all the functions called from the function). Function timing-abstractions have the following desirable effects:

- We can analyze a program and get approximate results even if the entire program is not available. In the case that a system designer has provided a time budget for certain software components, these budgets can be used to represent the execution time of the components until the components are complete. This allows the designer to time an incomplete program.
- It allows timing estimates to be made at an early stage in the development process.

8.3 Support for new Hardware Features

One important objective of WCET analysis research is to extend the set of hardware features that can be analyzed. A few years back, caches were considered intractable for

WCET analysis. Today we have analysis methods applicable to most common types of caches and single-pipeline processor cores. However, due to the rapid development in computer architecture, there is no end to complex hardware features that require tight modeling (or at least safe approximations).

8.4 Feedback into the Compiler

An intriguing idea is to use the results of the worst-case execution time analysis to guide the code generation in the compiler.

In [HSRW98], some synergy effects between the high-level analysis of loops and the compiler are shown. The detailed analysis of loops makes it possible to find redundant loop exit conditions (which can thus be removed as dead code). Furthermore, knowledge about the minimum number of loop iterations could help an optimizer make better decisions on how to optimize loops.

The analysis of the cache behavior of a program could be used to optimize the program for better cache behavior: either to make it more predictable, or to make it faster, depending on preference. In general, using the results of hardware analysis to obtain programs with more predictable timing behavior is a little-explored area which we believe should prove fruitful.

Another possibility is to use the identification of worst-case paths inherent in WCET analysis to optimize the speed of the worst-case execution. Today, compiler research focuses on speeding up the average case. However, for real-time systems, speeding up the worst case would be more appropriate, since systems are dimensioned for the worst case, and not for the average case.

8.5 Short Analysis Time or Accurate Result?

Early in a real-time development project, WCET analysis may be performed with the goal of obtaining approximate timing measures in order to make time-budgets or to estimate the level of CPU performance necessary. It is obvious that a very exact analysis is not needed at this stage, and that a general WCET tool should support "quick and dirty" analysis, where the goal is to quickly generate a WCET estimate, with no particular requirements on safety and tightness.

Thus, extensive analysis may not be needed until quite late in the development process, and the WCET tool should allow the programmer to choose the level of accuracy of the analysis. The final WCET estimate used to analyze and prove the shipping system must be safe and as tight as possible, of course.

9 Conclusions

In this article we have presented an architectural framework for Worst-Case Execution Time (WCET) analysis tools, which provides a flexible framework into which state of

the art WCET analysis techniques can be integrated. The architecture identifies the different subproblems associated with WCET analysis, and by providing well-defined data structures for the interfaces between the architectural components we allow modular upgrades/modifications of WCET tools built upon the architecture.

The main motivation for our work is that WCET analysis today is powerful enough to be exploited in industrial settings. However, WCET researchers have focused on smaller subproblems of WCET analysis, not considering the architectural issues of constructing a complete, industry strength, WCET tool.

Our architectural design decisions have been made to enable realization of the architecture in an industrial context. We believe that WCET analysis will be accepted by real-time practitioners only if it can be provided in the standard tool chest of an integrated development environment. WCET analysis should be a part of the everyday edit-compile-test-debug cycle.

Furthermore, we have to take the versatility of hardware being used for real-time systems into account. Many real-time systems are mass produced embedded systems, with extreme requirements on low hardware costs. Thus, real-time practitioners often target simple 8-bit architectures with very limited memory capacity.

However, we cannot limit a WCET tool to small CPUs but must also be able to support everything from the simplest hardware platforms up to full sized 32-bit RISC systems (i.e., systems with pipelines, multi-level caches, multi-issue, and other hardware features which substantially complicates the WCET analysis).

We have presented an overview of the state of the art in WCET analysis and shown how these techniques can be integrated into our framework. The ability to integrate results from many sources is important from at least two aspects: the practical aspect, that reusing work from other researches make a lot of sense (why reinvent the wheel?), and from the more academic perspective, that a framework into which many different methods can be joined makes it possible to make qualitative evaluations of different methods for solving a particular subproblem of WCET analysis (e.g., we could plug in two different cache analysis techniques and compare their effects on the final WCET estimate).

Acknowledgements

This work has been supported by ASTEC (Advanced Software Technology Center (`www.-docs.uu.se/astec`)), NUTEK (Swedish National Board for Industrial and Technical Development), TFR (Swedish Research Council for Energineering Sciences), and IAR Systems (`www.iar.com`).

We thank Tobias Amnell[9] and Jukka Mäki-Turja[10] for their comments on drafts of this article.

And by the way, the WCET estimate of the example program is 2040 clockcycles[11].

[9] Department of Computer Systems, Uppsala University

[10] Mälardalen Real-Time Research Center

[11] Calculated using Sicstus Prolog [Int95] Finite Domain Constraint Solver [COC97].

References

[ABD+95] N.C. Audsley, A. Burns, R.I. Davis, K. Tindell, and A.J. Wellings. Fixed Priority Pre-Emptive Scheduling: An Historical Perspective. *Real-Time Systems*, 8(2/3):129–154, 1995.

[Alt96] Peter Altenbernd. On the false path problem in hard real-time programs. In *Proc. of the 8th Euromicro Workshop of Real-Time Systems*, 1996.

[BCD+92] David S. Blickstein, Peter W. Craig, Caroline S. Davidson, R. Neil Faiman, Kent D. Glossop, Richard B. Grove, Steven O. Hobbs, and William B. Noyce. The gem optimizing compiler system. *Digital Technical Journal*, 4(4), 1992.

[BMSO+96] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding Instruction Cache Effects to Schedulability Analysis of Preemptive Real-Time Systems. In *Proc. 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 204–212. IEEE Computer Society Press, June 1996.

[CBW94] Roderick Chapman, Alan Burns, and Andy Wellings. Integrated program proof and worst-case timing analysis of SPARK Ada. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'94)*, 1994.

[Cha94] Roderick Chapman. Program timing analysis. Dependable Computing System Centre, University of York, England, May 1994.

[Cha95] R. Chapman. *Static Timing Analysis and Program Proof*. PhD thesis, Department of Computer Science, University of York, England, 1995.

[CO99] Mats Carlsson and Greger Ottosson. A comparison of cp, ip and hybrids for configuration problems. Technical Report T99-04, Swedish Institute of Computer Science, 1999.

[COC97] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In *Proc. 9th International Symposium on Programming Languages, Implementations, Logics, and Programs*, pages 191–206, 1997.

[CRTM98] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano – a revolution in on-board communications. *Volvo Technology Report*, 1:9–19, 1998.

[EAE98] Jakob Engblom, Peter Altenbernd, and Andreas Ermedahl. Facilitating worst-case execution times analysis for optimized code. In *Proc. of the 10th Euromicro Workshop of Real-Time Systems*, pages 146–153, June 1998.

[EG97] Andreas Ermedahl and Jan Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of Euro-Par 97, Lecture Notes in Computer Science (LNCS) 1300*, pages 1298–1307. Springer Verlag, August 1997.

[Eng99] Jakob Engblom. Static properties of embedded real-time programs, and their implications for worst-case execution time analysis. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*. IEEE Computer Society Press, June 1999. To be published.

[FMW97] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[GE98] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices*, 1998.

[HAM+99] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), January 1999.

[HP96] John L Hennessy and David A Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 2^{nd} edition, 1996. ISBN 1-55860-329-8.

[HSRW98] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proc. 4^{th} IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, June 1998. URL: `http://www.docs.uu.se/~mic/papers.html`.

[HW99] C. Healy and D. Whalley. Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints. In *Proc. 5^{th} IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, June 1999. To be published.

[Int94] Intel. *MCS 51 Microcontroller Family User's Manual*, 1994. Document no. 272383-002.

[Int95] Intelligent Systems Laboratory. SICStus Prolog user's manual. ISBN 91-630-3648-7, Swedish Institute of Computer Science, 1995.

[Jon99] Nigel Jones. Efficient c code for eight-bit mcus. *Embedded Systems Programming Europe*, pages 18–30, February 1999.

[LBJ+95] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An accurate worst-case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995. URL: `http://archi.snu.ac.kr/symin/ets.ps`.

[LHS+96] C. Lee, J. Han, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling. In *Proc. 17^{th} IEEE Real-Time Systems Symposium (RTSS'96)*, December 1996.

[LKM98] Sung-Soo Lim, Jihong Kim, and Sang Lyul Min. A worst case timing analysis technique for optimized programs. In *Proceedings of the fifth International Conference on Real-Time Computing Systems and Applications (RTCSA); Hiroshima, Japan*, pages 151–157, Oct 1998.

[LL73] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32:nd Design Automation Conference*, pages 456–461, 1995.

[LMW96] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modelling for real-time software: Beyond direct mapped instruction caches. In *Proc. 17th IEEE Real-Time Systems Symposium (RTSS'96)*, pages 254–263. IEEE Computer Society Press, December 1996.

[LS98] T. Lundqvist and P. Stenström. Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, June 1998.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design.* Morgan Kaufmann Publishers, 1997.

[Mue97a] Frank Mueller. Generalizing timing predictions to set-associative caches. In *Proc. of the 9th Euromicro Workshop of Real-Time Systems*, pages 64–71, Jun 1997. URL: `http://www.cs.fsu.edu/~mueller/publications.html`.

[Mue97b] Frank Mueller. Timing predictions for multi-level caches. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, pages 29–36, Jun 1997. URL: `http://www.cs.fsu.edu/~mueller/publications.html`.

[NEC95] NEC. *V850 Family 32/16-bit Single Chip Microcontroller User's Manual: Architecture*, 4th edition, 1995. Document no. U10243EJ4V0UM00.

[OS97] Greger Ottosson and Mikael Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, June 1997.

[PK89] Peter Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(1):159–176, 1989.

[PS90] Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on a source-level timing schema. In *Proc. 11th IEEE Real-Time Systems Symposium (RTSS'90)*, pages 72–81, December 1990.

[PS93] Peter Puschner and Anton Schedl. A tool for the computation of worst case task execution times. In *Proc. of the 5th Euromicro Workshop of Real-Time Systems*, 1993.

[PS95] Peter Puschner and Anton Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.

[SA97] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. Technical Report External Report 27-97, C-LAB, Paderborn, 1997.

[SKO+96] Veikko Seppänen, Anna-Maria Kähkönen, Markku Oivo, Harri Perunka, Pekka Isomursu, and Petri Pulli. Strategic needs and future trends of embedded software. Technical Report Technology Review 48/96, TEKES Technology Development Center, Oulu, Finland, October 1996.

[SSNB95] J.A. Stankovic, M. Spuri, M. Di Natale, and G.C. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, pages 16–25, June 1995.

[TF98] Henrik Theiling and Christian Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993. Out of print, but available from the author; see `http://cswww.essex.ac.uk/CSP/edward/FCS.html`.

[Vrc94] Alexander Vrchoticky. *The Basis for Static Execution Time Prediction.* PhD thesis, Institut für Technische Informatik, Technische Universitt Wien, Treitlstrae 3/182.1, A-1040 Wien, Austria, April 1994.

[Whi97] Randall T. White. *Bounding Worst-Case Data Cache Performance.* PhD thesis, Florida State University, 1997. URL: `http://www.cs.fsu.edu/~whalley/papers/white_diss97.ps`.

[WMH+97] R. White, F. Müller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *Proc. 3rd IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pages 192–202, June 1997.

[ZiL95] ZiLOG. *Z80 Microprocessor Family User's Manual.* ZiLOG, January 1995.