A Worst-Case Execution-Time Analysis Tool Prototype for Embedded Real-Time Systems[◇]

Jakob Engblom[*†]
IAR Systems AB
Box 23051, SE-750 23 Uppsala
Sweden
jakob.engblom@iar.se

Andreas Ermedahl[*]
IT-dept., Uppsala University
Box 325, SE-751 05 Uppsala
Sweden
andreas.ermedahl@docs.uu.se

Friedhelm Stappert[‡]
C-LAB
Fürstenallee 11, 33102 Paderborn
Germany
friedhelm.stappert@c-lab.de

Abstract

This paper gives an overview of the Worst-Case Execution Time (WCET) analysis research performed by the WCET group at Uppsala University, Sweden, in cooperation with C-LAB in Paderborn, Germany.

We have defined a modular architecture for a WCET estimation tool. The architecture has been used both to identify the components of the overall WCET analysis problem, and as a starting point for the development of a WCET tool prototype. Within this framework we have implemented and tested several modules for low-level timing analysis.

We have focussed on the needs of embedded real-time systems in designing our tools and directing our research. Our long-term goal is to provide WCET analysis as a part of the standard tool chain for embedded development (together with compilers, debuggers, and simulators). This is facilitated by our cooperation with the embedded systems programming-tools vendor IAR Systems.

Keywords: WCET analysis – software architecture – programming tools – embedded systems – hard real-time.

[◇] Presented at the RTTOOLS 2001 Workshop, Aalborg, Denmark, August 20, 2001

[*] This work is performed within the Advanced Software Technology (ASTEC, `http://www.docs.uu.se/astec`) competence center, supported by the Swedish National Board for Industrial and Technical Development (NUTEK, `http://www.nutek.se`).

[†] Jakob is an industrial PhD student at IAR Systems (`http://www.iar.com`) and Uppsala university, sharing his time between research and development work.

[‡] Friedhelm is a PhD student at C-LAB (`www.c-lab.de`), which is a cooperation of Paderborn University and Siemens.

1. Introduction

An increasing number of vehicles, appliances, power plants, etc. are controlled by computer systems interacting with their environments in real-time. 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. A common cause of failures is timing violations.

The purpose of *Worst-Case Execution Time* (WCET) analysis is to provide *a priori* information about the worst possible execution time of a piece of code before using it in a system.

WCET estimates are used in the development of real-time systems and embedded systems to perform scheduling and schedulability analysis, to determine whether performance goals are met for periodic tasks, to check that interrupts have sufficiently short reaction times, and for many other purposes.

To be valid, WCET estimates must be *safe*, i.e. guaranteed not to underestimate the execution time. To be useful, they must be *tight*, i.e. provide low overestimations. The safeness of an estimate is critical when the estimate is used in the construction of a safety-critical system.

The WCET of a piece of code depends both on the program flow (like loop iterations, decision statements, and function calls), and on architectural factors like pipelines and caches. Thus, both the program flow and the hardware the program runs on must be modeled by a WCET analysis method.

Our focus on embedded systems has guided us, both when deciding which types of hardware to focus on and to identify acceptable limitations regarding program flow and structure. We present the salient characteristics of the embedded systems field, and its repercussions on our work. Since the embedded marketplace is very fragmented we have strived to make our methods as portable and modular as possible.

A WCET tool should ideally be a component in an integrated development environment, making it a natural part of the embedded 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 [26], an embedded systems programming-tools vendor.

2. Background and Motivation

We begin by providing a background and motivation for our work, and the industrial context that we are considering.

2.1. Uses of WCET

The concept of a worst-case execution time for a program has been part of the real-time community for a long time, especially when doing schedulability analysis and scheduling [2, 4]. Many scheduling algorithms and all schedulability analysis assume knowledge about the worst-case timing of a task. However, WCET analysis have a much broader application domain; in any product development where timeliness is important, WCET analysis is a natural tool to apply.

Designing and verifying hard real-time systems (i.e. a system where a missed deadline is unacceptable) can be much simplified by using WCET analysis instead of extensive and expensive testing. WCET estimates can be used to verify that the response time of a critical piece of code is short enough, that interrupt handlers finish quickly enough, or that the sample rate of a control loop can be kept.

Tools for modeling and verification of real-time systems, like UppAal, HyTech, Kronos, and SPIN, can use WCET estimates to allow verification of actual implementations of systems.

When developing reactive systems using graphical programming tools like IAR visualSTATE, Telelogic Tau, and I-Logix StateMate, it is very helpful to get feedback on the timing for model actions and the worst-case time from input event to output event, as demonstrated by Erpenbach et al.[18].

WCET analysis can also be used to assist in selecting appropriate hardware. The designers of a system can take the application code they will use and perform WCET analysis for a range of target systems, selecting the cheapest (slowest) chip that meets the performance requirements.

For straight-line code, the WCET is the execution time for the code (assuming a predictable target platform). In this case, we can use WCET as the basis for programming tools to perform tricks like interleaving background tasks with a foreground program [10], or maintaining the timing of a virtual peripheral (i.e. a piece of software emulating a peripheral device)[1].

Chip Category	Number Sold
Embedded 4-bit	2000 million
Embedded 8-bit	4700 million
Embedded 16-bit	700 million
Embedded 32-bit	400 million
DSP	600 million
Desktop 32/64-bit	150 million

Figure 1. 1999 World Market for Microprocessors [50]

2.2. Target Hardware

In our work, we strive to provide tools that target the actual hardware and software used in embedded systems. This section discusses the hardware aspects of embedded systems, while the next section will discuss software.

An embedded system design is usually based on a *microcontroller*, i.e. a microprocessor and a set of peripherals integrated on the same die. Microcontrollers are packaged products like the Atmel AT90 line, or full-custom ASICs based on a standard CPU core (an example is Ericssons Bluetooth core using an ARM7).

As shown in Figure 1, microcontrollers completely outnumber the desktop chips in terms of units shipped. Only about 2% of the total number of chips used in 1999 were in desktop and server systems[2]. Also, simple microcontrollers dominate. The reason for this is that embedded systems designers, in order to minimize the power consumption, size, and cost of the overall system, use chips that are just fast and big enough to solve a problem.

For most 4-, 8-, and 16-bit processors, WCET analysis is a simple matter of counting the executing cycles for each instruction, since these CPUs are usually not pipelined. In our research, we have focussed on the need of the 32-bit and digital signal processors (DSPs).

Figure 2 shows market shares for 1999 in the 32-bit embedded processor segment. It is clear that rather simple architectures dominate the field. The best-selling 32-bit microcontroller family is the ARM from

[1]The use of software to replace hardware has grown very popular in the past few years, as exemplified by microcontroller concepts from Microchip, Scenix, Tera-Gen, and others.

[2]However, desktop processors represent a much larger share of the revenues, since the per-chip costs is in the order of dollars in the embedded field but in the order of hundreds of dollars in the desktop field.

Chip Family	Number Sold
ARM	151 million
Motorola 68k	94 million
MIPS	57 million
Hitachi SuperH	33 million
x86	29 million
PowerPC	10 million
Intel i960	7.5 million
SPARC	2.5 million
AMD 29k	2 million
Motorola M-Core	1.1 million

Figure 2. 1999 32-bit Microcontroller Sales [22]

Advanced Risc Machines [1]. All ARM variants have a single, simple pipeline, and very few have caches. The second-best selling architecture is the venerable Motorola 68k, which in most variants lack both pipeline and cache.

We conclude that our target hardware has the following characteristics:

- Pipelines are common on 32-bit chips, and they are usually scalar or VLIW. Out-of-order, dynamically scheduled pipelines are extremely rare.
- Floating-point pipelines or coprocessors are rare.
- Instruction caches are rare, (since they cost power and lead to performance which is harder to predict), and data caches are even more rare.
- On-chip RAM and ROM are the most important forms of memory, while external memory is avoided if possible due to cost and power considerations.
- The chip market is very fragmented, with tens of competing architectures just in the 32-bit field.

According to this, we have focussed on finding a WCET method that is easy to port and that supports the efficient handling of on-chip memory and peripherals, while allowing for the analysis of more advanced features in the future. Our first goal has been to handle scalar pipelines, and then expand to the more complex cases of superscalar architectures and caches.

2.3. Target Software

Since the WCET of a program depends heavily on the program flow, WCET analysis methods must be able to analyze and represent as much of the control flow of a program as possible.

Today, most embedded systems are programmed in C, C++, and assembly language [48]. More sophisticated languages, like Ada and Java, have found some use, but the need for speed, portability (there are C compilers for more architectures than any other programming language), small code size, and efficient access to hardware will keep C the dominant language for the foreseeable future.

We have investigated the properties of embedded software, used in actual commercial systems. The result is that while most of the code is quite simple (using singly nested loops, simple decision structures, etc.), there are some instances of highly complex control flow [11]. For instance, deeply-nested loops and decision structures do occur, and more problematically, recursion and unstructured code. Much of the complexity is due to automatically generated code, and since the number of code generators is expected to increase, the problems posed by generated code must be handled.

The most common focus for WCET analysis is user code, but in any system where an operating system is used, the timing of operating system services must also be taken into account. This means that WCET analysis must also consider operating system code. Colin and Puaut [8] have investigated parts of the code for the RTEMS operating system, and found no nested loops, unstructured code, or recursion. To the extent covered by their investigation, operating system code can be considered a well-behaved special case.

Ernst and Ye [17] reach some interesting conclusions regarding the actual program flow of some common signal-processing algorithms. While the program source code contains lots of decisions and loops, the decisions are structured in such a way that there is only a single path through the program – regardless of the input data. Identifying and expressing such single feasible paths is essential for tight WCET analysis.

Another important aspect of the expected software is that only small parts of the applications are really timing-critical. For example, in a GSM mobile phone, the GSM code is very small compared to the non-real-time user interface. Using this fact, a more ambitious WCET analysis can be performed on the timing-critical parts, provided that they can be efficiently identified.

Conclusions: We find it natural to support programs written in C, (and C++ in the future), possibly with inlined assembly code, and we need to have methods that efficiently handle nested loops, complex decision structures, recursion, and unstructured code. It should be possible to take advantage of detailed knowledge of the control flow, when such knowledge is available, and to analyze small parts of a large system (but taking the effects of the entire system into account).

2.4. Our Goals and Visions

Our long term goal is to produce a WCET tool, that is available as shrink-wrapped software. To this

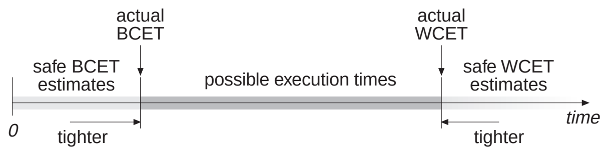


Figure 3. Relation between WCET, BCET, and Possible Program Execution Times

end, we need to perform basic research into particular methods, design a sound architecture for a tool, and find a way to come in contact with the development tools market.

On the research side, we are building a prototype system, integrating useful results from other researchers and filling in the missing gaps. We work from the low-level, basing our models on the hardware view of the software (i.e. the object code), since this is the only level at which all effects of the hardware and the programming environment is visible.

Regarding architecture, we are making the structure of the WCET tool as modular as possible, both to ease retargeting to new target hardware and to make it possible to use the components in other ways than just WCET analysis. For instance, flow analysis is useful for compiler optimizations.

On the industrial adaption end, we cooperate with IAR Systems (Uppsala, Sweden), a vendor of embedded system programming tools. We aim to integrate WCET analysis into their development products. WCET analysis is most appropriate as a new tool inside a familiar environment, not as a stand-alone tool.

We believe that this integration with accepted tools is the best way to get WCET analysis accepted on the market, and to make practitioners in the real-time field actually use execution-time analysis. To be really useful and to actually realize the potential benefits, WCET analysis should be employed on a daily basis.

3. WCET Analysis Overview and Related Work

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 a program (or program fragment). A related problem is that of finding the *Best-Case Execution Time* (BCET) of a program. See Figure 3 for an illustration of WCET, BCET, tightness, and safe estimates. Another execution time estimate is the *average* execution time, which is much harder to obtain analytically, since it requires statistical profiles of input data, instead of just boundary values, together with methods that can take advantage of such information.

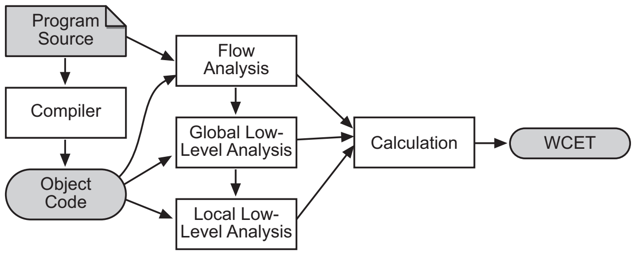


Figure 4. Components of WCET Analysis

When performing WCET analysis, it is assumed that there are no interfering background activities, such as direct memory access (DMA) or refresh of DRAM and that the program execution is uninterrupted (no preemptions or interrupts). Timing interference caused by such resource contention, for instance cache interference between tasks, should be handled by some subsequent analysis, e.g. schedulability analysis [3, 29, 46].

3.1. Main Components of WCET Analysis

To generate a WCET estimate, we consider a program to be processed through the following main steps:

- The *program flow analysis* calculates the possible flows through the program.
- The *low-level analysis* calculates the execution time for the instructions (in the detailed discussion below, low-level analysis is further divided into *global low-level analysis* and *local low-level analysis*).
- The *calculation* combines the above results into a WCET estimate.

Figure 4 shows the flow of information between these components. This structure serves as a conceptual classification of WCET research.

Program Flow Analysis

The task of program flow analysis is to determine the possible paths through a program, i.e. the dynamic behavior of the program. The result of the flow analysis is information about which functions get called, how many times loops iterate, if there are dependencies between different `if`-statements, etc. Since the problem is computationally intractable in the general case, a simpler, approximate analysis is normally performed. This analysis must yield safe path information, i.e. all feasible paths must always be covered by the approximation.

The flow information can be calculated manually, and communicated to the WCET tool by entering *manual annotations* into the program [43], or giving the flow information separately [20, 30, 41, 44].

Automatic flow analysis can be used to obtain flow information from the program source code without manual intervention [5, 6, 16, 24, 36, 49, 21].

Flow information can be generated on the source code or object code level. If generated on the source code level, the information must be mapped to the object code to be used in the WCET calculation. In the presence of optimizing compilers, this problem is non-trivial, since the flows of a program can be changed radically [12, 34].

Global Low-Level Analysis

The global low-level analysis considers the execution time effects of machine features that reach across the *entire program*. Examples of such features are *instruction caches*, *data caches*, *branch predictors*, and *translation lookaside buffers (TLBs)*. The analysis only determines how the global effects might affect the execution time, but does not generate actual execution times.

Since exact analysis is normally impossible, an approximate but safe analysis is necessary. For example, when an attempt is made to determine whether a certain instruction is in the cache, a cache miss is assumed unless we can be absolutely sure of a cache hit. This might be pessimistic but is definitely safe.

In WCET research, instruction caches [32, 31, 20, 40, 23, 49, 7], cache hierarchies [38], data caches [28, 40, 51, 49], and branch predictors [6] have been considered.

Local Low-Level Analysis

The local low-level analysis handles machine timing effects that depend on a single instruction and its immediate neighbors. Examples of such effects are *pipeline overlap* and *memory access speed*. On embedded systems, there are usually several different memory areas, each with different timing. On-chip ROM and RAM are fast, while off-chip memory typically takes several extra cycles to access.

Pessimistic (approximate but safe) approaches are common, e.g. assuming there is a pipeline speed-up effect only when enough pipeline content information is available to guarantee the effect.

Researchers have considered simple scalar pipelines [32, 40, 13, 23] and superscalar CPU pipelines [33, 47, 49].

Calculation

The purpose of the calculation is to calculate the WCET estimate for the program, given the program flow and global and local low-level analysis results. There are three main categories of calculation methods proposed in literature: *path-*, *tree-*, or *IPET- (Implicit Path Enumeration Technique)* based.

In a path-based calculation, the final WCET estimate is generated 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 represented *explicitly* [23, 49].

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 are used to make timing estimates for larger parts of the program [32, 6].

IPET-based methods express program flow and atomic execution times using algebraic and/or logical constraints. The WCET estimate is calculated by maximizing an objective function, while satisfying all constraints [30, 44, 40, 20].

Hybrid Approaches

There are some recent attempts to combine static WCET and measurement, to catch the "real" execution times, while trying to overcome the inherent problems of complexity and safety in measurement-based techniques.

Petters and Färber [42] perform sophisticated measurements of programs running on target hardware, aided by static off-line analysis. No attempt is made to perform a static time analysis, and the calculation is conceptually integrated with the cache and pipeline analysis.

Lindgren et al.[35] measure execution time and execution profiles of a program for a number of different inputs, and then try to determine the execution time for each independent path by solving the resulting system of linear equations. By ascribing a worst-case execution count to each path, they then calculate the worst-case execution time.

4. Prototype WCET System

Figure 5 gives an overview of our WCET analysis system as implemented today. In order to generate a WCET estimate, a program is processed through a number of steps (as described in Section 3 above).

There are two central data structures used in our tools: the *scope tree* and the *timing graph*. The scope tree reflects the structure of function calls and loops in the program, and is used for flow analysis and global low-level analysis. The timing graph represents an explicit low-level view of the program and is used for local low-level analysis.

The target chip for the present implementation is the NEC V850E, a typical 32-bit RISC microcontroller architecture [39]. It is mainly a classic RISC, but it has variable-length instructions to make the code smaller,

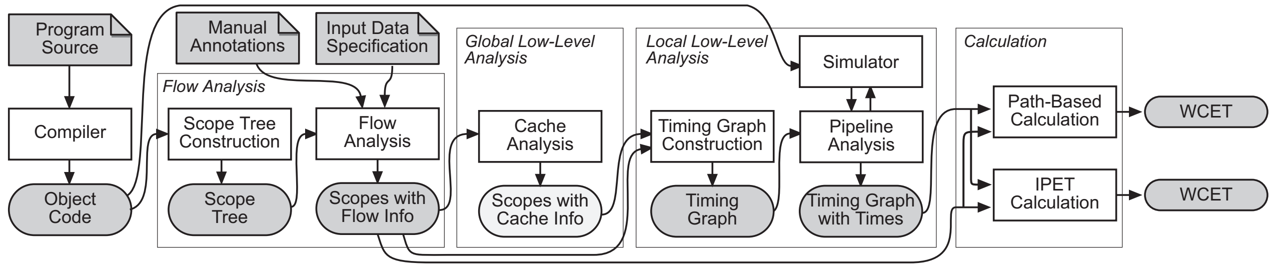
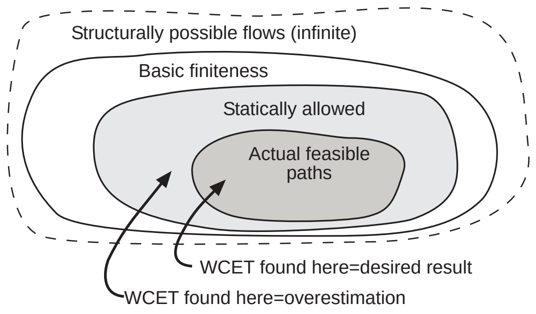


Figure 5. Detailed Overview of Our WCET System



Structurally possible flows (infinite)
Basic finiteness
Statically allowed
Actual feasible paths
WCET found here=desired result
WCET found here=overestimation

Figure 6. Effects of Flow Information

a pipeline that allows some instruction combinations to be issued on the same clock-cycle, and several complex instructions that address the typical embedded needs of bit manipulation and compact code. These features make it an appropriate vehicle for our experiments. Good contacts with NEC Europe also made information, support, and hardware available.

The compiler is a modified IAR V850/V850E C/Embedded C++ compiler [27] which emits the object code of the program in an accessible format. We only support C code in our prototype tool.

At present, each program must be contained in a single source file, since we need access to the whole program. In the future, we plan to integrate our tools with a whole-program compiler developed by the ASTEC WPO Project [45], which will remove the single-file limitation.

The simulator is a cycle-accurate model of the V850E [39], created in our research group using a homegrown generic framework for modeling pipelined processors.

At present, we have a prototype implementation of the low-level analysis: instruction cache analysis and pipeline analysis for the NEC V850E (based on the simulator). The prototype tool accepts input in the form of object-code files from the compiler and files containing flow information.

The calculation is either performed using IPET or Path-Based Techniques.

5. Flow Analysis

The set of *structurally possible flows* for a program, i.e. those given by the structure of the program, is usually infinite, since, e.g. loops can be taken an arbitrary number of times.

The executions are made finite by bounding all loops with some upper limit on the number of executions (*basic finiteness*). Adding even more information, e.g. about the input data, allows the set of executions to be narrowed down further, to a set of *statically allowed* paths. This is the "optimal" outcome of the flow analysis. Figure 6 provides an illustration of the different levels of approximation. Note that the set of actual feasible paths might be smaller than the statically allowed paths, due to approximations.

The task of the flow analysis is to identify the possible ways a program can execute. Deciding which of the possible paths that actually generate the worst execution time is done in the subsequent calculation phase.

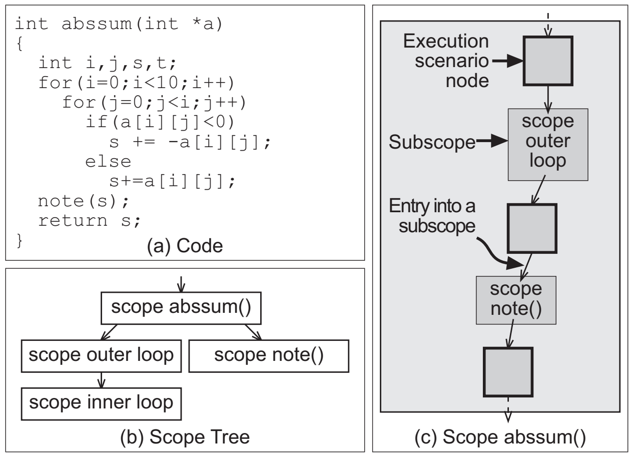
We therefore consider flow information handling to be divided into three phases:

1. **Flow information extraction**: Obtaining flow information by manual annotations or automatic flow analysis.
2. **Flow representation**: Representing the results of the flow analysis in a uniform manner.
3. **Conversion for calculation**: Converting the control flow (as represented in the flow representation) to a format suitable for the WCET calculation.

We believe that an interaction between manual annotations and automatic flow analysis is the best choice for flow information extraction. However, to avoid tedious work and errors from the programmer we should rely on automatic flow analysis as much as possible.

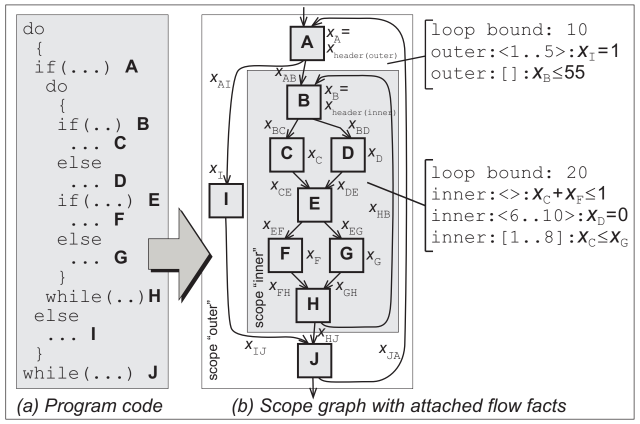
5.1. Representing Flow Information

To help us obtain tight WCET estimates, we have defined a flow representation formalism that is powerful enough to describe the complex flows found in embedded real-time systems (as discussed in Section 2.3). The representation is flexible enough to capture the output from a variety of flow analysis methods and



```
int abssum(int *a)
{
  int i,j,s,t;
  for(i=0;i<10;i++)
    for(j=0;j<i;j++)
      if(a[i][j]<0)
        s += -a[i][j];
      else
        s+=a[i][j];
  note(s);
  return s;
}
```

Figure 7. Example Scope Hierarchy



```
do
 {
 if(...) A
  do
   {
   if(..) B
    ... C
   else
    ... D
   if(...) E
    ... F
   else
    ... G
   }
  while(..)H
 else
  ... I
 }
while(...) J
```

Figure 8. Scopes with Attached Flow Facts

manual annotations, while remaining compact and efficient to work with [14].

The *scope graph* is a hierarchical representation of the dynamic structure of the program. Each scope corresponds to a certain repeating or differentiating execution context in the program, e.g. loops and function calls, and describes the execution of the object code of the program within that context.

Each scope is assumed to iterate, and has a header node. A maximal number of iterations must be given for each scope, and a new iteration is defined to start each time the header node is passed. Scopes are allowed to only iterate once, i.e. not loop.

An example of a small scope tree containing loops and function calls is shown in Figure 7[3].

Each scope has a set of associated *flow information facts*. Each flow information fact consists of three parts: the name of the *scope* where the fact is defined, a *context specifier*, and a *constraint expression*.

The fact is valid for *each entry* to the scope where it is attached. If the same scope is entered several times, each entry starts a new iteration count from zero.

The context specifier describes the iterations for which the constraint expression is valid. A context specification is either *total*, (written with "`[`" and "`]`"), for which the fact is considered as a sum over all iterations of the scope, or *foreach*, (written with "`<`" and "`>`"), which consider the fact as being local to a single iteration of the scope. Facts valid only for all iterations are expressed by `<>` or `[]`, while facts valid for certain iterations are expressed as `<`*min..max*`>` or `[`*min..max*`]` where *min* and *max* are integers and $min \leq max$.

The constraints are specified as a relation between two arithmetic expressions involving *execution count variables* and constants. An execution count variable, x_{entity}, corresponds to an entity in the scope graph (node or edge) and represents the number of times the entity is executed in the context given in the fact.

[3]The edges to and from the scopes in Figure 7(c) are only conceptual; the edges actually run between individual nodes in the scope.

In Figure 8 we show a number of flow information facts attached to two loop scopes `outer` and `inner`. Each scope has an upper loop bound attached to it, used for guaranteeing program termination.

The fact `inner : <> :` $x_{\mathtt{C}} + x_{\mathtt{F}} \leq 1$ means that the nodes `C` and `F` can never execute on the same iteration of the scope (an infeasible path), while the fact `inner : <6..10> :` $x_{\mathtt{D}} = 0$ gives that, for each entry of the inner loop, node `D` can not be executed during iterations 6 to 10.

The fact `inner : [1..8] :` $x_{\mathtt{C}} \leq x_{\mathtt{G}}$ gives that during the first eight iterations of `inner` node `C` can never be executed more times than the `G` node.

The fact `outer : <1..5> :` $x_{\mathtt{I}} = 1$ gives that for each entry of `outer`, during the first five iterations of `outer`, the execution is forced to take the path passing the `I` node, (and can therefore not enter `inner` during those iterations).

The fact `outer : [] :` $x_{\mathtt{B}} \leq 55$ is more complicated since its is attached to scope `outer` and reach into scope `inner` by referring to the header node of scope `inner`. It constrains the number of iterations of `inner` by specifying that, for each entry of scope `outer`, node `B` can not be executed more than 55 times.

The style of flow-limiting constraints used is well-known from the *implicit path-enumeration technique* (IPET) [30, 44, 40, 20]. The use of context specifications, however, makes this approach much more powerful than previous methods. For example, we are able to specify information locally in the scopes where it is valid. The unification of locally and globally valid information is unique. Also, information related to certain paths through a loop can be expressed using constraints on sequences of nodes.

We refer to [14] for a more detailed description of our flow information language. See also [14] and [19] on how to use flow information in an IPET-based respectively Path-based WCET calculation style.

6. Low-Level Analysis

The purpose of low-level analysis is to account for hardware effects on the execution time. As mentioned above, we consider low-level analysis to consist of two phases, the *global* effect analysis (handling hardware features that reach across the entire program) and the *local* effects analysis (handling hardware features that do not reach across the entire program).

6.1. Global Low-level Analysis

The global low-level analysis handles machine features that must be modeled over the *whole program* to be correctly analyzed. The global analysis determines facts that affect the execution time, but it does not generate concrete execution times. Examples of global effects are instruction caches, data caches, and branch predictors.

The results of the global low-level analysis are passed on to the local low-level analysis as *execution facts*. An execution fact tells whether a certain instruction hits or misses the instruction cache, whether a branch is correctly predicted or not, etc.

We use the execution facts to generate the correct execution times for the instructions of the program in the simulator. For example, the `icache` facts shown in Figure 9 are examples of the result of global low-level analysis.

We separate the global analysis from the local low-level analysis in order to get modularity, allowing global and local analysis methods to be mixed freely. Also, the execution fact approach allows several analyses to be performed on the same program, with cumulative results. For example, instruction cache and branch prediction analysis could be performed separately, making each analysis simpler.

6.2. Local Low-level Analysis

The local effects analysis handles machine timing effects that depend on a *single instruction and its immediate neighbours*. Examples of local effects are pipeline overlap between instructions and basic blocks, memory speed (particularly important for embedded real-time systems, where multiple memory banks with different speeds are common), and instruction alignment (on our example CPU, the NEC V850E, 32-bit instructions not aligned on a 32-bit boundary may take longer time to execute).

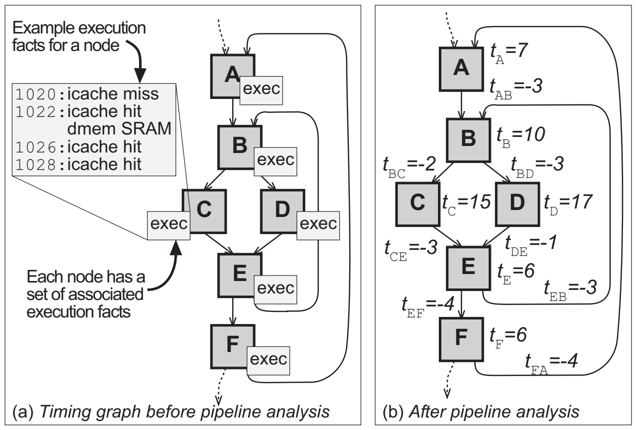


Figure 9. Example of Timing Graph

The low-level analysis operates on a *timing graph*, which is a flat graph for the entire program. The nodes and edges in the timing graph correspond to the nodes and edges in the scope tree, but the scope structure is removed, since it is not relevant at this level.

Each node in the timing graph has an associated execution scenario, generated in the global low-level analysis and during the timing graph construction (where issues like memory access time and branch taken/not taken are converted to execution facts).

Figure 9(a) shows an example of a timing graph. The example execution facts shown indicate whether instructions hit or miss the instruction cache (`icache hit` and `icache miss`), and the memory type accessed by a load instruction (`dmem SRAM`). Each scope with a copy of the same basic block could give a different set of facts (that is one of the purposes of scopes).

6.3. Extracting Pipeline Effects by Simulation

The primary problem 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 [32, 40, 49].

We solve this problem in a novel and portable way by using a *simulator* to obtain execution times for timing graph nodes and sequences of nodes. The simulator takes instructions together with the execution facts (to determine the execution for instructions with varying behavior).

The pipeline analysis generates times for the nodes and edges in the timing graph. Times for nodes correspond to the execution times of basic blocks (with the associated execution scenarios) in isolation, (e.g. $t_{\mathtt{Q}}$), and times for edges, (e.g. $t_{\mathtt{QR}}$), to the pipeline effect when the two successive nodes are executed in sequence (usually an overlap).

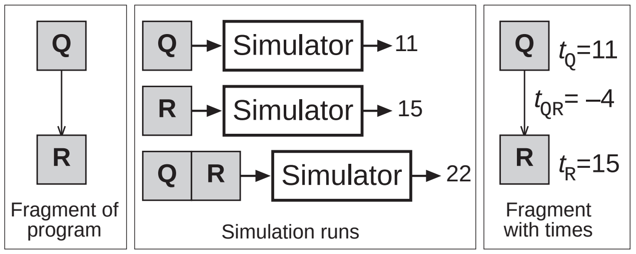


Figure 10. Timing Effect Calculation

Timing effects for sequences of nodes are calculated by first running the individual nodes in the simulator, and then the sequence and comparing the execution times. The process is illustrated in Figure 10. The timing effect for the edge `QR` is $22 - 15 - 11 = -4$; the time is negative since the two nodes `Q` and `R` overlap. The details of these issues, and how to handle pipeline timing effects appearing over sequences of nodes longer than two, are given in [13].

Simulators are a standard part of embedded development environments today, and are often provided by the chip manufacturers. We expect to utilize this fact to quickly port our pipeline analysis to new chips.

7. Calculation

The purpose of the *Calculation* phase is to calculate the final WCET estimate for the program.

We have investigated two calculation methods, namely the Implicit Path-Enumeration Technique (IPET), and a Path-Based Method.

7.1. IPET-Based Calculation Method

The IPET for WCET analysis was introduced by Puschner and Schedl [44]. A similar approach was presented by Li and Malik [30]. We have extended IPET to be able to handle more advanced CPU architectures [40, 13] and more complicated flows [14].

In IPET, the flow of a program is modeled by execution count variables. The values reflect the total number of executions of each node or edge for an execution of the program.

The input to the calculation phase is a timing graph where each entity (nodes, edges, or sequences of nodes), have corresponding execution count and time variables (called x_{entity} and t_{entity}).

The value of an execution count variable corresponds to the number of times the entity is executed, and the time variable gives the contribution of that entity to the total execution time each time it is executed.

The WCET estimate is generated by maximizing the sum of the products of the execution counts and execution times (subject to the flow constraints):

$$WCET = maximize(\sum_{\forall entity} x_{entity} \cdot t_{entity})$$

This maximization problem is solved using a constraint solver or integer linear programming (ILP) system.

Observe that IPET will not explicitly find the worst case execution *path*, i.e. the precise order in which all nodes are executed, since paths are not explicitly represented. However, the execution counts can be interpreted as an *execution count profile* of the worst-case execution, which is very useful to identify hot spots and bottlenecks in the program.

7.2. Path-Based Calculation Method

In a *path-based approach*, the possible execution paths of a program or piece of a program are explicitly explored to find the longest path [23, 25, 49]. In contrast to IPET, the path-based approach explicitly computes the longest executable path in the program. This may be valuable information for the programmer, e.g. for tuning and debugging purposes. Previous path-based approaches have handled flow information by explicitly extracting and representing all possible paths within a program, and then removing infeasible paths from the WCET calculation.

The classic approach to longest executable path search using path-based calculation is to generate all possible paths for a certain program segment (function, loop body, or other unit), run all the paths through some kind of hardware model, and select the path with the longest execution time. The unit of analysis is the complete path, and the number of paths to explore is up to 2^n, where n is the number of decisions in the program segment being analyzed.

The need to handle complete paths arises from the use of pipelining in modern processors: to get a tight timing estimate, one must account for the overlap between basic blocks, and this can only be done by analyzing all the basic blocks in a path in a continuous sequence.[4]

However, since our pipeline analysis allows the timing of a path to be composed from smaller components, it is possible to reformulate the longest path search problem as finding the longest path in a directed acyclic graph. This can be solved very efficiently using well-known basic techniques, like Dijkstra's Algorithm [9].

We extended the basic algorithm to handle flow information, expressed using the flow language described above. Furthermore, we extended the algorithm to

[4]To keep complexity under control while losing some precision, it is possible cut a program segment into smaller pieces with a lower number of decisions in each [23].

handle arbitrary pipeline effects, going beyond pipeline effects between adjacent basic blocks. For a more detailed description we refer to [19].

We are currently investigating whether it is possible to create a hybrid approach between the path-based and IPET-based calculations, combining the efficiency of path-based approaches with the expressive power of IPET.

8. Validating WCET Methods and Tools

For a WCET tool or method implementation to be used in the development of a safety critical system we must be able to guarantee that it produces safe and reasonably tight results.

When evaluating WCET analysis methods, the common methodology is to compare a WCET estimate with an execution of the same program with known worst-case data on the target hardware. This evaluation method is problematic, since it mixes the effects of several sources of errors. For example, a pessimistic hardware model might mask errors in a flow analysis that generates too short program paths – the resulting estimates might appear to be safe for any given set of test cases, but there could be cases where the analysis would be unsafe. Also, if errors are detected, it is very hard to pinpoint the error source.

According to Section 3 above, we consider WCET analysis to be divided into several independent components. It is necessary to consider the correctness (and effectiveness) of each component in isolation, since otherwise errors in one component may mask errors in other components. Each component must be safe and tight in its own right in order for the complete analysis system to be safe and tight.

In [15], we applied the idea of component-wise isolation and testing to our tool in order to give evidence that the *pipeline analysis* and *calculation method* we are using are safe and tight.

In [37], we investigated and improved the quality of our V850E simulator to more accurately reflect the real chip. The result was a simulator which in most cases is precise vs. the hardware, with deviations on the order of a few percent overall.

The ability to validate each component in isolation is an important advantage of our modular approach to WCET analysis. Of course, it is also very important to show that we preserve the safeness when combining the different components, e.g. the results of the cache and pipeline analysis, in the complete WCET tool.

9. Example Analysis Results

To illustrate the precision achievable with our tool, Figure 11 shows some measurements comparing the analysis results with actual execution times. The calculation method used is path-based, and the target CPU is the NEC V850E (updated simulator version, giving actual execution times different from [14]).

The column *Basic* gives the WCET estimate using only loop-bounds as flow limiting information and ignoring pipeline overlap *between* nodes (but including the pipeline overlap *within* nodes[5]). Columns including *Flow* hold WCET estimates resulting from adding flow facts to the programs. Columns including *Pipeline* hold WCET estimates where pipeline effects both within and between nodes have been accounted for. *Actual* gives the actual WCET of the program, as given by a simulation of the target platform. The numbers in the *+%* columns give the pessimism of each WCET estimate in percent.

The worse results for the columns without pipeline show that the modeling of pipelines is very important for tight WCET analysis. In most cases, the effect of the pipeline is much larger than that of the control flow. The results for including flow show that in some instances, the number of statically allowed flows (given by the facts) is larger than the actual possible flows, but that it is usually possible to get very close to the actual WCET.

The `insertsort` program has a type of flows that is not very suitable for the path-based calculation, but which is handled well by the IPET calculation method [14]. This points to the possibility of using different calculation methods for different programs.

10. Conclusions and Future Work

In this paper we have described the motivation, strategy and achievements of our WCET group.

We are working with a focus on embedded real-time systems, which affects certain demands on the methods and tools we develop, and affects the priorities assigned to various components of the WCET analysis problem.

We are still working on the tool and the methods needed to produce a fully working industrial-strength tool. Our plan is to reuse as much previous research as possible, while filling in the gaps between previous methods and making sure that integration works. The following are some of the topics that we will investigate in the near future:

Our aim is to analyze full C, and to remove academic assumptions, e.g. we will allow unstructured code. The flow analysis will be made at the intermediate code level. In this way, we catch (most) compiler optimizations. Furthermore, the approach depends less on the

[5]Completely ignoring pipeline effects within a block would create a WCET about five times higher (since our chip has a five-stage pipeline).

	Basic		With Flow		With Pipeline		Flow & Pipeline		Actual
Program	**Cycles**	**+%**	**Cycles**	**+%**	**Cycles**	**+%**	**Cycles**	**+%**	**Cycles**
compress	126242	+1357	10388	+20	92482	+967	8672	+0.12	8662
crc	61624	+104	61624	+104	30389	+0.39	30389	+0.39	30271
expint	68077	+693	10062	+17.2	41359	+382	8588	0	8588
fibcall	559	+78.6	559	+78.6	313	0	313	0	313
fir	487970	+40.2	487808	+40.1	352162	+1.2	352073	+1.1	348095
insertsort	2328	+117	2328	+117	1794	+67.0	1794	+67.0	1249
jfdctint	5388	+9.4	5388	+9.4	4942	+0.35	4942	+0.35	4925
lcdnum	501	+153	341	+72.2	283	+42.9	198	0	198
matmult	278859	+24.4	278859	+24.4	221824	0	221824	0	221824
ns	22903	+64.6	20653	+48.5	15434	+10.9	13934	+0.2	13911
nsichneu	150841	+195	87193	+70.6	97662	+91	51133	+0.03	51116

Figure 11. Execution Time Estimates (Path-Based)

source programming language, making it easier to port to other languages.

In the real world, there will be parts of a program that are not available as source code. Thus, we need to find ways to handle incomplete programs, standard libraries, and operating system interfaces in WCET analysis. Work on this will be performed in cooperation with the whole-program compilation group in Uppsala [45].

References

[1] ARM (Advanced Risc Machines) WWW Homepage. URL: `http://www.arm.com`.

[2] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Fixed priority pre-emptive scheduling: an historical perspective. *Real-Time Systems*, 8(2/3):129–154, 1995.

[3] 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.

[4] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano – A Revolution in On-Board Communications. *Volvo Technology Report*, 1:9–19, 1998.

[5] R. Chapman, A. Burns, and A. 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.

[6] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Journal of Real-Time Systems*, May 2000.

[7] A. Colin and I. Puaut. A Modular and Retargetable Framework for Tree-Based WCET Analysis. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*, June 2001.

[8] A. Colin and I. Puaut. Worst-Case Execution Time Analysis for the RTEMS Real-Time Operating System. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*, June 2001.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[10] A. Dean and J. P. Shen. System-Level Issues for Software Thread Integration: Guest Triggering and Host Selection. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS'99)*, 1999.

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

[12] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating Worst-Case Execution Times Analysis for Optimized Code. In *Proc. 10th Euromicro Workshop of Real-Time Systems*, pages 146–153, June 1998.

[13] J. Engblom and A. Ermedahl. Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99).* IEEE Computer Society Press, December 1999.

[14] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS'00)*, pages 163–174, November 2000.

[15] J. Engblom and A. Ermedahl. Validating a Worst-Case Execution Time Analysis Method for an Embedded Processor. In *Work-in-Progress Session, 21st IEEE Real-Time Systems Symposium (RTSS'00)*, November 2000.

[16] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. 3rd International European Conference on Parallel Processing, (Euro-Par'97), LNCS 1300*, pages 1298–1307, August 1997.

[17] R. Ernst and W. Ye. Embedded Program Timing Analysis Based on Path Clustering and Architecture Classification. In *International Conference on Computer-Aided Design (ICCAD '97)*, 1997.

[18] E. Erpenbach, F. Stappert, and J. Stroop. Compilation and Timing Analysis of Statecharts Models for Embedded Systems. In *Proc. 2nd International Workshop on Compiler and Architecture Support for Embedded Systems, (CASES'99)*, October 1999.

[19] F. Stappert and A. Ermedahl and J. Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. Technical Report 12, Dept. of Information Technology, Uppsala University, 2001. `http://www.it.uu.se/research/reports/2001-012/`.

[20] C. Ferdinand, F. Martin, and R. 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.

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

[22] T. R. Halfhill. Embedded Market Breaks New Ground. *Microprocessor Report, January 17*, 2000.

[23] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1), January 1999.

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

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

[26] IAR Systems WWW Homepage.
URL: `http://www.iar.com`.

[27] IAR Systems. *V850 C/EC++ Compiler Programming Guide*, 1st edition, January 1999.

[28] S.-K. Kim, S. L. Min, and R. Ha. Efficient Worst Case Timing Analysis of Data Caching. In *Proc. 2nd IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 230–240, 1996.

[29] 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. 17th IEEE Real-Time Systems Symposium (RTSS'96)*, December 1996.

[30] Y-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proc. of the 32:nd Design Automation Conference*, pages 456–461, 1995.

[31] Y-T. S. Li, S. Malik, and A. 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.

[32] 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.

[33] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Multiple-Issue Machines. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998.

[34] S-S. Lim, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Optimized Programs. In *Proc. 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98)*, pages 151–157, Oct 1998.

[35] M. Lindgren. Measurements and Simulation Based Techniques for Real-Time Systems Analysis. Licentiate Thesis, Uppsala University of Technology, Uppsala, Sweden, 2000.

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

[37] Sven Montán. Validation of Cycle-Accurate CPU Simulator against Actual Hardware. Master's thesis, Dept. of Information Technology, Uppsala University, 2000. Technical Report 2001-007, `http://www.it.uu.se/research/reports/2001-007/`.

[38] F. Müller. 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.

[39] NEC Corporation. *V850E/MS1 32/16-bit Single Chip Microcontroller: Architecture*, 3rd edition, January 1999. Document no. U12197EJ3V0UM00.

[40] G. Ottosson and M. 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.

[41] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62, March 1993.

[42] S. Petters and G. Färber. Making Worst-Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, December 1999.

[43] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *The Journal of Real-Time Systems*, 1(1):159–176, 1989.

[44] P. Puschner and A. Schedl. Computing Maximum Task Execution Times with Linear Programming Techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.

[45] J. Runeson. Code compression through procedural abstraction before register allocation. Master's thesis, Department of Information Technology, Uppsala University, March 2000.

[46] J. Schneider. Cache and Pipeline Sensitive Fixed Priority Scheduling for Preemptive Real-Time Systems. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS'00)*, pages 195–204, nov 2000.

[47] J. Schneider and C. Ferdinand. Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, May 1999.

[48] V. Seppänen, A-M Kähkönen, M. Oivo, H. Perunka, P. Isomursu, and P. Pulli. Strategic Needs and Future Trends of Embedded Software. Technical Report Technology Review 48/96, TEKES Technology Development Center, Oulu, Finland, October 1996.

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

[50] David Tennenhouse (Intel Director of Research). Keynote Speech at the 20th IEEE Real-Time Systems Symposium (RTSS'99), Phoenix, Arizona, December 1999.

[51] 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.