Worst-Case Execution-Time Analysis for Embedded Real-Time Systems*

Jakob Engblom[§†], Andreas Ermedahl[§], Mikael Sjödin[§‡]
www.docs.uu.se/{~jakob,~ebbe,~mic}

Jan Gustafsson[¶], Hans Hansson[§¶]
www.idt.mdh.se/personal/{jgn,han}

[§]*Department of Computer Systems, Uppsala University, Sweden*
[¶]*Mälardalen Real-Time Research Centre, Sweden*

Abstract

In this article we give an overview of the Worst-Case Execution Time (WCET) analysis research performed by the WCET group of the ASTEC Competence Center at Uppsala University.

The basis for this work is our modular architecture for a WCET tool, used both to identify the components of the overall WCET analysis problem, and as a starting point for the development of an industry strength WCET tool prototype. Within this framework we have proposed solutions to several key problems in WCET analysis, including representation and analysis of the control flow of programs, modeling of the behavior and timing of pipelines and other low-level timing aspects, integration of the control flow information and low-level timing to obtain a safe and tight WCET estimate, and validation of our tools and methods.

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 substantially facilitated by our close cooperation with the embedded systems programming-tools vendor IAR Systems.

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

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. A common cause of such failures is timing problems.

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.

In this article, we present the work we have performed on WCET analysis. We present both the methods developed and a WCET tool framework in which they are used.

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 caches and pipelines. Thus, both the program flow and the hardware the program runs on must be modeled by a WCET analysis method.

* The 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.

‡ Supported by the Swedish Research Council for Energineering Sciences (TFR, http://www.tfr.se).

Our focus on embedded systems have 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 are today. 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 [IARa], an embedded systems programming-tools vendor.

The main technical contributions of our work are:

- A modular architecture for a WCET tool, where different WCET analysis components can be incorporated.
- Two control flow analysis methods, one based on abstract interpretation [EG97, Gus00] and one based on structural analysis of loops [HSRW98, HSR+00].
- A compact and efficient method for representing information about the control flow of a program [EE00a].
- A pipeline analysis method that uses a generic CPU simulator instead of a special-purpose WCET CPU model [EE99].
- A calculation method that allows control flow and hardware analysis results (including the effects of caches) to be integrated and used to efficiently calculate tight and safe WCET estimates [OS97, EE99, EE00a].
- A method for validating the components of our WCET tool, aiming at a complete validation of the entire tool suite [EE00b].
- We have investigated the properties of commercial embedded real-time programs and the attitudes of real-time practitioners regarding WCET tools and WCET analysis in general [Eng99, Gus00].

Paper outline: In Section 2 we give a motivation for WCET analysis with focus on embedded real-time system development. Section 3 presents previous work and Section 4 gives an overview of our WCET tool. Sections 5, 6, and 7 present our work on the different components of WCET analysis. In Section 8, we outline work performed on validation of WCET tools. Section 9 gives experimental results. Finally, Section 10 presents conclusions and ideas for future work.

2. Background and Motivation

We begin by providing a background and motivation for our work, in 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 [ABD+95, CRTM98]. Many scheduling algorithms and all schedulability analysis assume some form of knowledge about the worst-case timing of a task. However, we consider WCET analysis to have a much broader application area. 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 [BLL+98], SPIN [Hol97], HyTech [HHWT97], and Kronos [BDM+98] can use WCET analysis to obtain the timing of the code in a system, allowing verification to be applied to implementations as well as models of systems.

When developing reactive systems using graphical programming tools like IAR visualSTATE [IARb], Telelogic Tau [Tel], and I-Logix StateMate [I-L], 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. [ESS99].

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 (assuming that there are usable models of the target systems available).

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 [DS99], or maintaining the timing of a virtual peripheral (i.e. a piece of software emulating a peripheral device)[1].

[1] The use of software to replace hardware has grown very pop-

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 [Ten99]

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 the software.

Embedded system designs are usually based on *microcontrollers*, microprocessors with 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 were in desktop and server systems.[2] Also, simple microcontrollers dominate. The reason for this is that embedded systems designers use chips that are just fast and big enough to solve a problem, in order to minimize the power consumption, size, and cost of the overall system.

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

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 Advanced Risc Machines [ARM]. 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 pipelines and cache.

ular in the past few years, as exemplified by microcontrollers from Scenix [Sce00], Tera-Gen [Mic99], and others.

[2]Note however, that the desktop processors represent a much larger share of the revenues, since the per-chip costs is on the order of dollars in the embedded field but 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 [Hal00]

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 unpredictable performance), 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 [SKO+96]. More sophisticated languages, like Ada and Java, have found some use, but the need for speed, portability, 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, to provide some data about the types of program flows to expect. The result is that while most of

the code is quite simple (using single-nested loops, simple decision structures, etc.), there are some instances of highly complex control flow [Eng99].

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 code generators are expected to grow in use over the next years, 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 [CP99] have investigated how the code for the RTEMS operating system is written, and found no nested loops, unstructured code, or recursion. In this case, operating system code can be considered a well-behaved special case.

Ernst and Ye [EY97] 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 mobile phone, the GSM code is very small compared to the non-real-time user interface. Thus, the WCET analysis can be rather computation-intensive, provided that the timing-critical parts can be efficiently extracted.

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. Users of Execution Time Analysis

The expected users of a WCET tool are system designers and programmers (both for basic system software and for application software).

In order to find out more about the need for WCET analysis in an industrial environment, an enquiry was sent out to Swedish companies related to real-time and embedded systems during the spring of 1997. The answers contained a lot of interesting information about current practice of WCET estimation.

WCET analysis was used to verify real-time requirements, to optimize programs, to compare algorithms, and to evaluate hardware. None of the companies used a commercial WCET tool. Among the many measurement tools used were emulators, time-accurate simulators, logic analyzers and oscilloscopes, timer readings inserted into the software, and software profiling tools.

Practically everyone answered that a WCET tool would be valuable, for the following reasons:

- to save time, as measurements become unnecessary.
- it would be advantageous to be able to see the time directly for the program being developed.
- the tool could show execution times for different processors, clock frequencies, etc., without running the program on these targets.

The following functions in a WCET tool were regarded as valuable by a clear majority of the answers:

- mean execution times, as well as minimum and maximum execution times.
- input data dependency.
- supervision of time budgets, (a time budget is an upper limit of the WCET of a program, calculated e.g. in the system design phase).
- specification of the input data for which the program exceeds its time budget.
- identification of the parts of the program that execute for certain input.
- hypothetical execution times for empty code blocks, i.e., a possibility to reserve slots with a hypothetical execution time for code to be developed later.
- a choice between a fast but coarse analysis, and a slow but more exact one.
- WCET calculation between timer points in the code.

The inquiry was followed up by interviews with selected companies, as summarized in [Gus00].

2.5. Our Goals and Visions

Our long term goal is to produce a WCET tool, that is available as shrink-wrapped software. To this 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 gaps we find. We work from

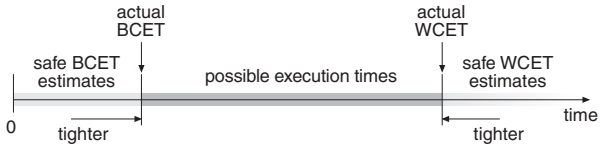


Figure 3. The Relation between WCET, BCET, and Possible Program Execution Times.

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 integrated development environment. WCET analysis is most appropriate as a new tool inside a familiar environment, not as a standalone tool.

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

3. WCET Analysis Overview and Previous 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.

When performing WCET analysis, it is assumed 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 DRAM. Timing interference caused by such resource contention should be handled by some subsequent analysis, for instance schedulability analysis [BMSO+96, LHS+96].

To generate a WCET estimate, we consider a program to be processed through a number of steps: *program flow analysis*, *low-level analysis*, and *calculation*. The low-level analysis is further divided into *global low-level analysis* and *local low-level analysis*.

Program Flow Analysis

The task of the 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 on which functions get called, how many times loops iterate, if there are dependencies between different `if`-statements, etc.

The information can be obtained using *manual annotations* integrated in the programming language [Par93, PK89], or as additional information [FMW97, LM95, PS95]). *Automatic flow analysis* can also be used to obtain the flow information from the program source code without manual intervention [CBW94, EG97, HSRW98, LG98, LS98, SA00, Gus00].

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 factors are instruction caches, data caches, branch predictors, and translation lookaside buffers (TLBs). The analysis only determines how global effects affect the execution time but it does not generate actual execution times.

For WCET analysis, instruction caches [LBJ+95, FMW97, HAM+99, SA00], cache hierarchies [Mül97], data caches [KMH96, WMH+97, SA00], and branch predictors [CP00] 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.

Researchers have considered simple scalar pipelines [LBJ+95, EE99, HAM+99] and superscalar CPU pipelines [LHKM98, SF99, SA00].

Calculation Method

The purpose of the *calculation* is to calculate the final 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.

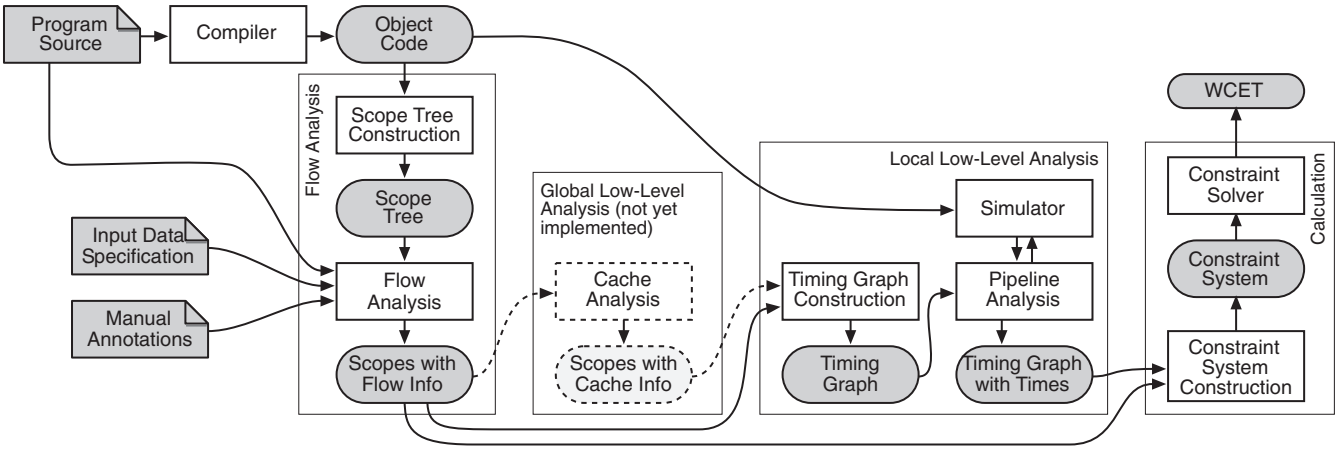


Figure 4. Overview of our current WCET analysis system

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 *explicitly* represented [HAM+99, SA00].

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 [LBJ+95, CP00].

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 [LM95, PS95, OS97, FMW97].

Integrated Approaches

Most WCET tools integrate several of the above steps into a single tool, even though the algorithms are kept separate. There are also some methods that integrate several steps into a single algorithm, making the above division inappropriate.

Lundqvist and Stenström [LS98] use a modified CPU simulator to simultaneously perform flow, cache, and pipeline analysis, and calculation. The results are nice but rely on a very sophisticated and detailed CPU modeling, and the method is therefore not very portable.

Liu and Gomez [LG98] as well as Persson and Hedin [PH98] perform WCET analysis on the source code of a program by assigning timing values to operations of the target language (Scheme respectively Java). Such analysis is not effective for programs running on modern hardware or compiled using optimizing compilers, since the time for each operation will in these cases depend on its context. This is similar to the problems encountered with the early timing schema approach [PS90].

Bernat et. al performs WCET analysis on the Java Byte code [BBW00] level. Their method is easily portable between different target platforms but encounters similar effectivity problems as the source code WCET analysis methods mentioned above.

Petters and Färber [PF99] 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.

4. Prototype WCET System

Figure 4 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). Note that although the global low-level analysis is currently not implemented, the implementation is prepared to include this in a future.

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 [Cor99]. It is mainly a classic RISC, but it has variable-length instructions to make the code

smaller, 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++ [IAR99] compiler 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 with a whole-program compiler developed by the ASTEC WPO Project [Run00], which will remove the single-file limitation.

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

In the following sections, we will present the research we have performed on each problem. Note that the two flow analysis methods are very different in character and in the form of information generated. We are exploring the tradeoffs between analysis power and cost, both in terms of calculation and data representation.

Such trade-offs and the use of varying representations for analysis and analysis results is very common in compiler research and construction, since different representations are appropriate for different types of analysis.

The results can in all cases be converted to our unifying notation for WCET flow information [EE00a].

5. Flow Analysis

The task of the flow analysis is to identify the possible ways a program can execute. The final WCET path extraction is made 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 final 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.

Our WCET group has developed methods for automatic flow analysis [EG97, Gus00, HSRW98, HSR+00]. We have developed a framework for representing obtained flow information and described how the flow information can be converted to the IPET style of calculation [EE00a]. We have also investigated how to convert flow information obtained on the source code level to the object code level, in the presence of optimizing compilers [EAE98].

5.1. Automatic flow analysis based on abstract interpretation

In [EG97, Gus00] we present a flow analysis method based on abstract interpretation [Cou96]. The idea is to extract properties of the run-time behavior of a program by making an "interpretation" of the program using abstractions of values instead of concrete values. Based on the data abstraction we must also make a transformation of the constructs in the target language, from a concrete semantics handling concrete values to an abstract semantics running on the abstract values.

The goal is to obtain a safe approximation of the possible executions of the program, i.e. the calculated set of paths must be equal to, or a superset of, the set of possible paths. For example, it is allowed to report that the program iterates more times in a loop than it actually does, but not the other way around. The safety of the analysis can be proven using abstract interpretation theory. Provided that the abstraction of values and the semantic operations in the language are safe, we can show that the approximation of the run-time behavior of a analyzed program is also safe.

The results of the analysis are safe estimations of the number of iteration of loops and the possible outcome of decision statements (in some relation to the iterations of surrounding loops). As a side-effect, safe bounds of variables' values at different points in the program are also generated.

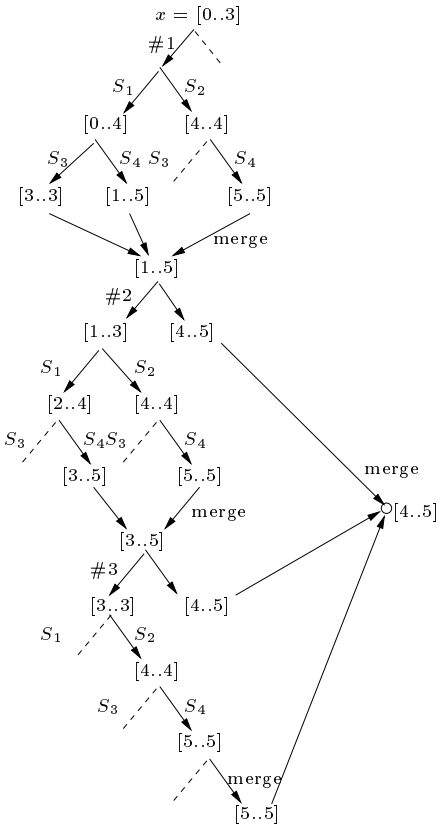
The current analysis is performed at the source code level of a program, but the same ideas should be applicable to intermediate and object code analysis. The abstract domain for variable values currently implemented is intervals and split intervals, but more complex value domains might be considered in the future.

As an example of the type of flow information that can be generated, consider the code fragment in Figure 5(a). We will analyze the program using our automatic flow analysis method, using intervals as the abstract domain.

Figure 5(b) shows the structure of the analysis. In

```
/* Input limits for x: 0 ≤ x ≤ 3 */
while(x < 4) {
  if (x < 3) x = x * 2;     S1
  else x = x + 1;           S2
  if (x == 1) x = x + 2;    S3
  else x = x + 1;           S4
}
```

(a) Example program.



(b) Flow analysis run.

Figure 5. Example Program and Flow Analysis

the figure, the analysis starts at the top with the initial value for x which is the interval [0..3]. As the while-loop cannot terminate at this point for any possible input value, the analysis continues in iteration #1. The infeasible loop termination path is indicated with a dashed line.

Since both edges are feasible in the first if-statement, the analysis continues in both. In S_1, x will assume the value [0..2] initially and [0..4] at the end of the edge. In S_2, the corresponding values will be [3..3] and [4..4].

Two possible x values will encounter the second if-statement. For the first, x = [0..4], both S_3 and S_4 are feasible, leading to two possible end cases where x is [3..3] and [1..5], respectively. In the second case, S_3 is infeasible, since $x \neq 1$, i.e. we have found the infeasible path $S_2 \rightarrow S_3$, marked with a dashed line in the figure. Thus, only S_4 is analysed further, with an end value x = [5..5].

When the analysis of the first iteration of the loop is finished our method *merges* the values to form the union of all variable values. The purpose of this is to reduce the complexity of the calculation. In the figure, we see that the merged value of x is [1..5].

The analysis of iterations #2 and #3 are performed similar to #1. We can see that four additional infeasible paths are found. All possible exit values of x are merged together to form the final value of x.

To sum up, the flow analysis extracted the following information: the loop may iterate one to three times, and the path $S_2 \rightarrow S_3$ can never be executed. Furthermore, in iteration two the path $S_1 \rightarrow S_3$ is impossible, and the statement S_1 cannot be executed in the third iteration. Safe bounds on possible values for x where also generated.

Executing the program for all possible inputs, (possible for this small input space), we can see that the flow analysis result is safe but somewhat pessimistic, since the path $S_1 \rightarrow S_3$ is also infeasible in the first iteration, something not discovered by our analysis.

5.2. Bounding the Number of Loop Iterations

The single most important piece of flow information is the maximum number of iterations performed in each loop. Hence, methods to automatically derive tight bounds on the number of loop iterations are of great importance.

In cooperation with the WCET research group at Florida State University (Florida, USA), we have developed a method to predict tight bounds on a multitude of different types of loops [HSRW98, HSR+00]. The method can bound the number of iterations of, for instance,

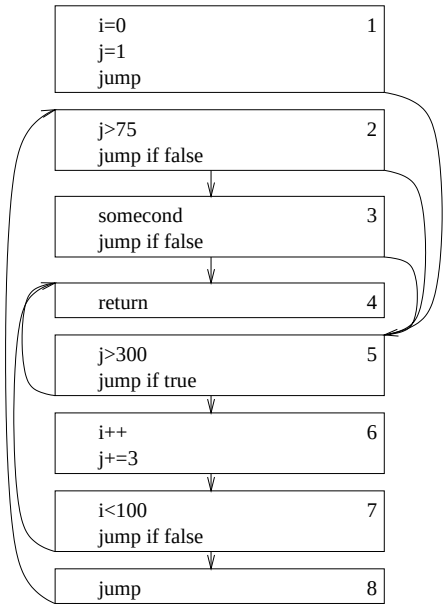
- loops with multiple exits,
- loops with multiple loop-index variables,
- loops that have bounds that are dependent on variables or function parameters (assuming that the values of the variables can be predicted, e.g. using our abstract interpretation method), and
- nested loops where the number of iterations of the inner loop depends on the loop index of the outer loop.

The method works on optimized object code and is thus language independent. It is currently implemented in an optimizing C compiler for the SPARC architecture [BD88], but our intentions are to implement it in our current framework.

For an example on how the method works, consider the C-program in figure 6(a) and its compiled counter-

```
main()
{
  int i,j;
  extern int somecond;
  for(i = 0, j = 1; i < 100; i++, j+=3)
    if(j > 75 && somecond || j > 300)
      break;
}
```

(a) Source code.



(b) Compiled pseudo assembler.

Figure 6. Example Program for Loop Bound Analysis.

part in figure 6(b). The details of the method are described in [HSRW98, HSR+00].

The method progresses in four phases:

1. The branches that can affect the number of loop iterations are iteratively identified. First, we identify branches that either exit the loop or that restart another iteration. Second, we find the branches that indirectly affect the number of iterations by conditionally transferring control to one of two branches already identified.
   In figure 6, we have identified all branches affecting the number of loop iterations.
2. Next, branches whose direction depends on loop-index variables are identified and the iteration number when they change direction is determined.
   In figure 6 the branches in blocks 2, 5 and 7 are identified as being dependent on loop-index variables and they change direction on iteration number 26, 101 and 101 respectively.
3. Using the information from step 2 we can find out during which iterations it is possible to reach the branches identified in step 1 (i.e. branches that exit the loop or continue to the next iteration).
4. Finally, knowing when different branches can be reached and in which direction they will jump we can calculate the minimum and maximum number of iterations of the loop.
   For the example in figure 6 the minimum number of iterations is 26 and the maximum is 101.
   As a side effect we also identify the test in block 5 to be redundant and hence block 5 could be removed by a dead-code elimination pass in the compiler.

In the case where loop bounds depend on variables the method produces a symbolic expression including the dependent variables. Given bounds on the values of those variables the number of iterations can be calculated. This, for instance, allows different bounds to be calculated for different calls to functions that have loops that are dependent on function parameters.

For nested loops the symbolic expression may contain loop-index variables from an outer loop. In those cases we consider the loop-index variable value for each iteration of the outer loop, thus obtaining a tight estimate of the number of iterations of the inner loop.

5.3. 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 above). The representation is flexible enough to capture the output from a variety of flow analysis methods and manual annotations, while remaining compact and efficient to work with [EE00a].

Our representation is based on the concept of a *scope*. Each scope corresponds to a certain repeating or differentiating execution context in the program (typically a function call or loop, but the exact definition is up to the flow analysis method employed), and describes the execution of the object code of the program within that context.

Since the behavior in a scope may depend on the call path to the scope (e.g. a function called with different parameters from different places in a program), we use the *scope tree* structure to describe the dynamic execution of the program[3]. Each function or loop can be present more than once in the tree, corresponding to different execution contexts.

[3]Note that the compiler concept of a call graph [Muc97] bundles different invocations of a function into a single node, which would give a loss in precision for WCET analysis.

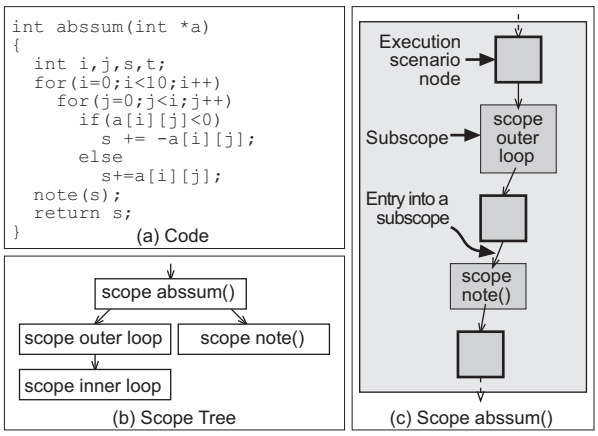


Figure 7. Example Scope Hierarchy

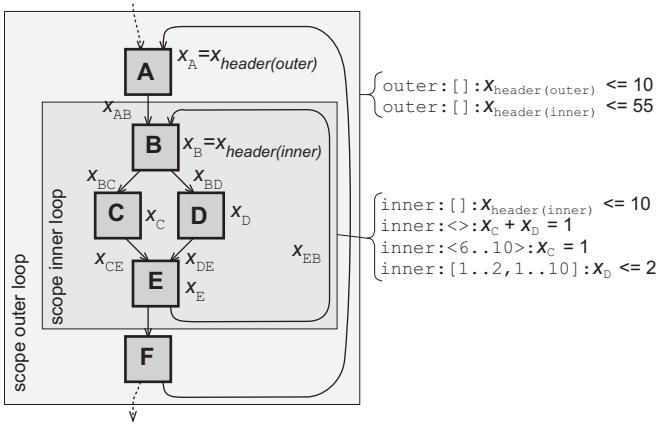


Figure 8. Example of Scopes with Attached Flow Facts

A structure similar to our scope tree is used in most other WCET work [FMW97, PK89, HAM+99, CP00]. An example of a small scope tree containing loops and function calls is shown in Figure 7.

A scope consists of *nodes* and *edges*. A node belongs to exactly one scope, and represents the execution of a certain basic block of the program in the environment given by a scope and its superscopes.

Inside each scope, we have nodes corresponding to the execution of a certain basic block of the program in the context given by the scope. Note that although we are discussing program flow, we are doing so on a structure given by the object code of the program. Therefore, the flow analysis information must be mapped onto the object-code structure of a program.

All scopes are supposed to be looping, thus there exists a concept of *iterations* of scopes (note that recursive functions are considered the same as loops). Each scope has a *header node* which has the property that no other node in the scope can be executed more than once without every possible execution path passing the header node. The iteration numbers of a scope can be thought of as being counters which are reset to zero at the entry to scopes and incremented by one each time the header node is executed.

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 context specifier describes the iterations for which the constraint expression is valid. This can either be for all iterations or for certain iterations. The type of a context specification is either *total*, (within "[" and "]"), for which the fact are considered as a sum over all iterations of the specified scopes, or *foreach*, (within "<" and ">"), which consider the fact as being local to a single iteration of the scope. Facts valid for all iterations are expressed by <> or [], while facts valid for certain iterations are expressed as <*ranges*...> or [*ranges*...]. A range specification can cover several scopes, e.g. [1..2,1..10].

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 like a node, a sequence of nodes or an edge.

In Figure 8 we show a number of flow information facts attached to the two loop scopes:

- A simple loop bound is specified by using an all/total operator on the count variable that corresponds to the header node in the loop: $inner : [] : x_{header(inner)} \leq 10$.
- A loop bound for one scope can also be specified for each entry of an upper scope, e.g. the $outer : [] : x_{header(inner)} \leq 55$ fact constrains the iteration of the inner loop more than the local loop bound does.
- $inner : <> : x_{\mathtt{C}} + x_{\mathtt{D}} = 1$ The nodes C and D cannot execute on the same iteration of the scope.
- $inner : <6..10> : x_{\mathtt{C}} = 1$ Node C must execute on all the iterations numbered 6 thru 10.
- $inner : [1..2,1..10] : x_{\mathtt{D}} \leq 2$ Node D cannot be executed more than twice in the given iteration space (covering two scopes).

The style of constraints used is well-known from the *implicit path-enumeration technique* (IPET) [LM95, PS95, OS97, FMW97]. 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 cer-

tain paths through a loop can be expressed using constraints on sequences of nodes.

Our flow information language is strong enough to handle the flow information generated by existing program analysis methods, while providing compact representations even for complex flows.

We refer to [EE00a] for a more detailed description of our flow information language, and how locally specified flow information can be converted to global constraints that are used for our IPET style of WCET calculation, (see Section 7).

5.4. Problems with Optimizing Compilers

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 [EAE98, LKM98].

We have performed some research on the problem. The solution we devised was to describe the effect that each compiler transformation has on the flow information, and have the compiler emit a trace of the transformations made during optimization of the program. The transformation trace is then used to transform the corresponding flow information [Eng97, EAE98].

An alternative solution to this problem is to perform flow analysis on the intermediate code level of the compiler. The intermediate code level contains enough information from the source code to get the intention of the programmer and is close enough to the object code to have been subjected to most of the optimizations in the compiler. We plan to port our flow analysis methods to the intermediate code format used by the IAR System C-compilers.

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.

The execution facts are used 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.

An example of cache analysis that fits very well with our approach is the VIVU instruction cache analysis performed by Ferdinand et al. [FMW97]. Here, each instruction is given a hit or miss classification, which is trivial to express as execution facts. Note that the scope tree might have its structure changed in the process of the global analysis. The VIVU approach splits each loop scope into two scopes, one for the first and one for the other iterations.

We split the global analysis from the local low-level analysis in order to get modularity, allowing different global analyses to be used with the same local analysis, and vice versa. Also, the execution fact approach allows several analyses to be performed on the same program, with cumulative results. For example, instruction cache and branch predicition analysis could be performed separately, making each analysis simpler.

We have not yet implemented a cache analysis in our prototype tool, since caches are not very common on our target systems (see Section 2.2 above).

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 V850, 32-bit instructions not aligned on a 32-bit boundary may take longer time to execute).

The low-level analysis operates over 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

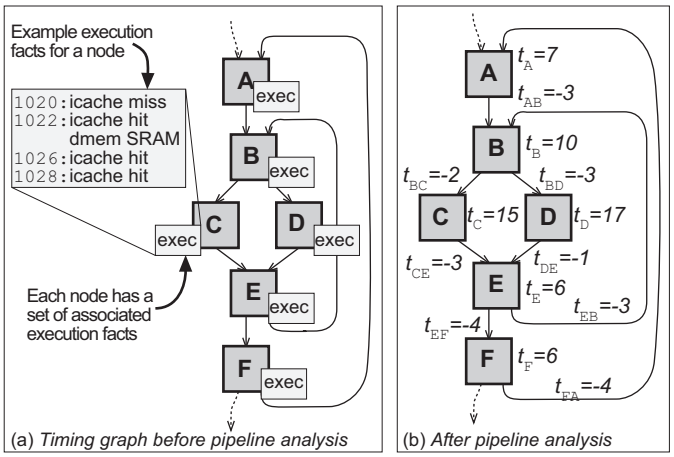


Figure 9. Example of Timing Graph

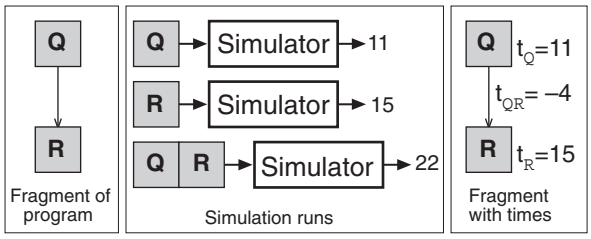


Figure 10. Timing Effect Calculation

taken are converted to execution facts). Figure 9(a) shows an example of a timing graph.

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 [LBJ+95, OS97, SA00].

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).

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.

Since pipeline effects can potentially appear across sequences of nodes longer than two, we run progressively longer sequences of nodes until a *termination condition* is satisfied, generating new times, $t_{sequence}$, as needed. The termination condition depends on the CPU used, and is true when there is no possibility for a longer sequence to have any effect on the execution time of the program. The details of these issues are given in [EE99].

Simulators are a standard part of embedded development environments today, and are often provided by the chip manufacturers. We expect to utilise 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 chosen IPET as our calculation method since it is the method that best satisfies our goal of modularity, and since it allows for the expression of the most complex flows (including unstructured flow). We believe that path- and tree-based methods are harder to retarget to new platforms and yield more problems in integrating results from different program flow and low-level analyses.

The IPET method for WCET analysis was introduced by Puschner and Schedl [PS95]. A similar approach was presented by Li and Malik [LM95]. We have extended IPET to be able to handle more advanced CPU architectures [OS97, EE99].

In IPET, the flow of a program is modeled as an assignment of values to execution count variables. The variables are considered *global*, and the values reflect the total number of executions of each node for each execution of the program.

The input of 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 part of the program 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. At present, we use the constraint solver of the SICStus Prolog system [ISL95].

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.

7.1. Constraint Generation

The WCET expression requires that we have constraints on all execution count variables. We distinguish between three types of execution count constraints: *structural constraints*, *finiteness and start constraints*, and *tightening constraints*.

The possible flow given the structure of the program is modeled using structural constraints. For each node in the timing graph, the sum of the incoming flows is equal to the outgoing flows. For example, for node `B` in Figure 8 the constraints $x_{\mathtt{AB}} + x_{\mathtt{EB}} = x_{\mathtt{B}}$ and $x_{\mathtt{B}} = x_{\mathtt{BC}} + x_{\mathtt{BD}}$ would be generated.

For sequences longer than two nodes, representing long pipeline overlaps or specific paths in the program, the generation of structural constraints is a little bit more complicated, but can be handled, as explained in [EE99]. Our tool automatically generates the structural constraints from the timing graph.

In addition to the structural constraints, we need constraints to state that the program is finite, and that it executes once. The *basic finiteness* of a program is ensured by constraining each loop or recursive function by an upper bound, stating the maximal number of iterations of the loop. The program is set to execute once by constraining the execution count of the entry node of the program to one (i.e. $x_{entrynode} = 1$).

Going beyond basic finiteness, we may add flow information facts to further tighten the execution time estimates. They are not necessary to obtain a WCET estimate, but enhance the quality of the estimate.

Generating the basic finiteness and the tightening flow information facts is the responsibility of the flow analysis. In [EE00a] we describe how basic finiteness and tightening constraints, provided in our flow information representation (see Section 5.3), can be converted to a format suitable for our IPET-style of calculation.

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. In [EE00b] we address the problems faced during systematic testing of WCET analysis methods.

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 [EE00b], 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.

We also show the correctness of our local and global low-level analysis algorithms, independent of the quality of the simulator used. The next step is to show the correctness of the simulator in itself, relative to the target hardware.

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

Finally, we will present some analysis results obtained using our prototype tool. We have used a number of benchmarks used by other WCET groups, and a few programs of our own making. A summary of the programs is given in Figure 11.

The execution times in our experiments are shown in Figure 12. The column *Basic* gives the WCET estimate using only basic finiteness constraints. The column *Improved* gives the estimate resulting from adding

Program	Description	Properties
`fibcall`	Simple iterative fibonacci calculation, used to calculate fib(30).	Parameter-dependent function, single-nested loop.
`matmult`	Matrix multiplication of two 20x20 matrices.	Multiple calls to the same function, nested function calls, triple-nested loops.
`jfdctint`	Discrete-cosine transformation on a 8x8 pixel block.	Long calculation sequences (i.e. long basic blocks), single-nested loops.
`fir`	Finite impulse response filter (signal processing algorithms) over a 700 items long sample.	Inner loop with varying number of iterations, loop-iteration dependent decisions.
`crc`	Cyclic redundancy check computation on 40 bytes of data.	Complex loops, lots of decisions, loop bounds depend on function arguments, function that executes differently the first time it is called.
`insertsort`	Insertion sort on a reversed array of size 10.	Input-data dependent nested loop with worst-case of $n^2/2$ iterations.
`duff`	Using "Duff's device" [Ray00] to copy a 43 byte array.	Unstructured loop with known bound, jump table for `switch` statement.

Figure 11. Benchmark programs

	Basic		Improved		No pipeline		Actual
Program	Cycles	+%	Cycles	+%	Cycles	+%	Cycles
`fibcall`	287	0.3	286	0.0	616	115.4	286
`matmult`	239528	0.0	239528	0.0	312047	30.3	239528
`jfdctint`	5550	0.0	5550	0.0	6197	11.7	5550
`fir`	326967	1.1	323277	0.0	505276	56.3	323277
`insertsort`	2077	66.3	1249	0.0	1435	14.9	1249
`duff`	1248	1.8	1226	0.0	2116	72.6	1226
`crc`	35878	7.0	35436	5.7	71918	114.6	33518

Figure 12. Execution Time Estimates

flow information facts. The column *No pipeline* gives the WCET estimate obtained when assuming the flow information and pipeline overlap *within* nodes but ignoring the pipeline overlap *between* nodes. The column *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 better results for the "improved" column indicate the advantage of using more complex flows than simple loop bounds in WCET analysis, and show that our method is able to capture complex flows in an effective and efficient manner. The sensitivity to flow analysis depends on the program.

One should also note that for some programs, simple loop bounds are sufficient to describe the behavior, while other programs have very hard-to-describe flows (for example `crc`, where we were unable to capture all the wrinkles of the flow).

The much worse results for the "no pipeline" column show that the modeling of pipelines is very important for tight WCET analysis. In most cases, the effect of the pipeline is much greater than the control flow. Note that we are still using pipeline effects inside each basic block – completely ignoring pipelines would create a WCET about five times higher (our chip has a five-state pipeline).

It is obvious that the length of the basic blocks vary between the programs: for `jfdctint`, we have few and long blocks, giving a low overestimation without pipelines, while `crc` and `fibcall` have many short blocks and are very sensitive to pipeline analysis.

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 puts certain demands on the methods and tools we develop, and affects the priorities assigned to various components of the WCET analysis problem.

Our approach is based on the IPET- (Implicit Path Enumeration Technique) method [PS95] for calculating worst-case execution times, since it is the most flexible and powerful calculation method we have found.

We presented the following main contributions of our group:

- A modular architecture for a WCET tool, where different WCET analysis components can be incorporated. The architecture allows integration of previous published components, thus we leverage on state-of-the art methods, and we have a platform that allows us to compare different methods to implement a particular component. We have implemented a straw-man prototype of the architecture, and several of the components.
- Two control flow analysis methods, one based on abstract interpretation [EG97, Gus00] and one based on structural analysis of loops [HSRW98, HSR+00].
- A compact and efficient method for representing information about the flow of a program [EE00a], including an algorithm to convert the flow information for use in an IPET calculation.
- A pipeline analysis method that uses a generic CPU Simulator instead of a special-purpose WCET CPU model [EE99]. The advantage of this approach is that the simulator can be verified in isolation and that the WCET analysis method is easy to port to new architectures.
- A calculation method that allows flow and hardware analysis results (including the effects of caches) to be integrated and used to efficiently calculate tight and safe WCET estimates [OS97, EE99, EE00a].
- A methodology for validating the components of our WCET tool, aiming to obtain a complete validation of the entire tool suite [EE00b]. We have validated the pipeline analysis and calculation method, and are working on the present simulator.
- Our work on investigating the properties of real programs and the wishes of real-time programmers regarding WCET tool features [Eng99, Gus00].

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:

We plan to investigate how cache and branch-prediction analysis can take advantage of flow information. This should allow us to provide tighter estimates than previous methods.

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. This work is performed in cooperation with the whole-program compilation group in Uppsala [Run00].

We are investigating the use of our pipeline analysis method for VLIW and in-order superscalar pipelines, since this would allow us to perform WCET for DSP processors.

In a long-term perspective, we need to investigate several issues involved in the industrial deployment of WCET tools. There is, for instance, a need for a good graphical user interface to the WCET analysis tool, both to direct the WCET analysis to relevant program parts and to present the results of the analysis. The integration of manual annotation and automatic flow analysis into the compiler used is another very important practicality issue.

Acknowledgements

We would like to thank Bengt Jonsson and Friedhelm Stappert for their fruitful comments on drafts of this article.

References

[ABD+95] 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.

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

[BBW00] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis using Java Byte Code. In *Proc. of the 12th Euromicro Workshop of Real-Time Systems*, pages 81–88, 2000.

[BD88] M. E. Benitez and J. W. Davidson. A Portable Global Optimizer and Linker. In *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pages 329–338, June 1988.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. KRONOS: A Model-Checking Tool for Real-Time Systems. In *Proc. of the 10th International Conference on Computer Aided Verification*, pages 546–550. Springer-Verlag, 1998. LNCS 1427.

[BLL+98] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New Generation of UPPAAL. In *Proceedings of the International Workshop on Software Tools for Technology Transfer, Aalborg, Denmark*, July 1998.

[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] 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.

[Cor99] NEC Corporation. *V850E/MS1 32/16-bit Single Chip Microcontroller: Architecture*, 3^{rd} edition, January 1999. Document no. U12197EJ3V0UM00.

[Cou96] Patrick Cousot. Abstract Interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.

[CP99] A. Colin and I. Puaut. Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System. Technical Report No 1277 (Publication Interne), IRISA, November 1999.

[CP00] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Journal of Real-Time Systems*, May 2000.

[CRTM98] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano – A Revolution in On-Board Communications. *Volvo Technology Report*, 1:9–19, 1998.

[DS99] A. Dean and J. P. Shen. System-Level Issues for Software Thread Integration: Guest Triggering and Host Selection. In *Proc. 20^{th} IEEE Real-Time Systems Symposium (RTSS'99)*, 1999.

[EAE98] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating Worst-Case Execution Times Analysis for Optimized Code. In *Proc. of the 10^{th} Euromicro Workshop of Real-Time Systems*, pages 146–153, June 1998.

[EE99] J. Engblom and A. Ermedahl. Pipeline Timing Analysis Using a Trace-Driven Simulator. In *Proc. 6^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, December 1999.

[EE00a] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21^{st} IEEE Real-Time Systems Symposium (RTSS'00)*, November 2000. Accepted for publication.

[EE00b] J. Engblom and A. Ermedahl. Validating a Worst-Case Execution Time Analysis Method for an Embedded Processor. In *Proc. 21^{st} IEEE Real-Time Systems Symposium (RTSS'00)*, November 2000.

[EG97] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer Verlag, August 1997.

[Eng97] J. Engblom. Worst-Case Execution Time Analysis for Optimized Code. Master's thesis, Department of Computer Systems, Uppsala University, September 1997. DoCS MSc Thesis 97/94.

[Eng99] J. Engblom. Static Properties of Embedded Real-Time Programs, and Their Implications for Worst-Case Execution Time Analysis. In *Proc. 5^{th} IEEE Real-Time Technology and Applications Symposium (RTAS'99)*. IEEE Computer Society Press, June 1999.

[ESS99] E. Erpenbach, F. Stappert, and J. Stroop. Compilation and Timing Analysis of Statecharts Models for Embedded Systems. In *In The Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99), Washington, D.C*, October 1999.

[EY97] 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.

[FMW97] 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.

[Gus00] J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation.* PhD thesis, Department of Computer Systems, Information Technology, Uppsala University, May 2000.

[Hal00] T. R. Halfhill. Embedded Market Breaks New Ground. *Microprocessor Report, January 17*, 2000.

[HAM+99] 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.

[HHWT97] T. A. Henzinger, P-H. Ho, and H. Wong-Toi. HYTECH: A Model Checker for Hybrid Systems. In *Proc. of the 9^{th} International Conference on Computer Aided Verification*, pages 460–463, 1997. LNCS 1254.

[Hol97] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[HSR+00] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting Timing Analysis by Automatic Bounding of Loop Iterations. *Journal of Real-Time Systems*, May 2000.

[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.

[I-L] I-Logix WWW Homepage.
URL: `http://www.ilogix.com/smover_c.htm`.

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

[IARb] IAR Systems. Reference Applications of VisualSTATE.
`http://www.iar.dk/products/references.htm`.

[IAR99] IAR Systems. *V850 C/EC++ Compiler Programming Guide*, 1^{st} edition, January 1999.

[ISL95] ISL (Intelligent Systems Laboratory). SICStus Prolog user's manual. ISBN 91-630-3648-7, Swedish Institute of Computer Science, 1995.

[KMH96] S.-K. Kim, S. L. Min, and R. Ha. Efficient Worst Case Timing Analysis of Data Caching. In *Proc. of RTAS'96* , pages 230–240. IEEE, 1996.

[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.

[LG98] Y. A. Liu and G. Gomez. Automatic Accurate Time-Bound Analysis for High-Level Languages. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'98)*, 1998.

[LHKM98] 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.

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

[LKM98] S-S. Lim, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Optimized Programs. In *Proc. of the fifth International Conference on Real-Time Computing Systems and Applications (RTCSA); Hiroshima, Japan*, pages 151–157, Oct 1998.

[LM95] 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.

[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.

[Mic99] Microprocessor Report. Tera-Gen Reveals 8-bit Threaded Processor. *Microprocessor Report, January 25*, 1999.

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

[Mül97] 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.

[OS97] 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.

[Par93] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62, March 1993.

[PF99] 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.

[PH98] P. Persson and G. Hedin. Interactive Execution Time Predictions using Reference Attributed Grammars. In *Proc. of the 2:nd Workshop on Attribute Grammars and their Applications (WAGA'99), Amsterdam, Netherlands*, pages 173–184, Aug 1998. URL: `http://www.dna.lth.se/home/Patrik_Persson`.

[PK89] 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.

[PS90] C. Y. Park and A. 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.

[PS95] 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.

[Ray00] E. Raymond. The Jargon File, version 4.2.0. `http://www.tuxedo.org/~esr/jargon/html/index.html`, February 2000.

[Run00] J. Runeson. Code compression through procedural abstraction before register allocation. Master's thesis, Department of Information Technology, Uppsala University, March 2000.

[SA00] 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.

[Sce00] Scenix Semiconductor Inc. *Scenix SX Family User's Manual*, 3rd edition, 2000.

[SF99] J. Schneider and C. Ferdinand. Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation. In *Proc. SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*. ACM, May 1999.

[SKO+96] 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.

[Tel] Telelogic WWW Homepage. URL: `http://www.telelogic.com/tau4`.

[Ten99] David Tennenhouse (Intel Director of Research). Keynote Speech at the 20th IEEE Real-Time Systems Symposium (RTSS'99), Phoenix, Arizona, December 1999.

[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.