Chapter 2
WCET Analysis Techniques

Contents

2.1 Introduction

Besides functional correctness, hard real-time systems have to satisfy timing constraints. They can be validated by a schedulability analysis that tests whether all tasks of the real-time system executed on the given hardware satisfy their timing constraints. The schedulability analysis requires safe and precise bounds on the execution time of each task. In this chapter, common timing analysis techniques found in academia and industry are discussed. Section 2.2 introduces the three main classes of approaches for a WCET analysis. The static timing analysis is the only method that guarantees safe WCET estimates. It does not rely on the execution of the code under analysis but derives its timing information from the program code combined with abstract models of the hardware architecture. Section 2.3 defines basic terms involved in the static analysis. Finally, the workflow of the leading static WCET analyzer *aiT* [Abs10], which is also exploited in this work, is presented in Sect. 2.4.

P. Lokuciejewski, P. Marwedel, *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems,* Embedded Systems,
DOI 10.1007/978-90-481-9929-7_2, © Springer Science+Business Media B.V. 2011

2.2 Approaches for WCET Analysis

The general classification of the approaches for a WCET estimation is based on the distinction whether the considered task is executed or statically analyzed.

2.2.1 Measurement-Based Approach

This method executes the task on the given hardware or in a simulator and measures the execution time. Representative sets of input data are provided which are supposed to cover scenarios where the maximal program execution time can be measured. This method has two main disadvantages.

First, it is not safe since the input to the program leading to the worst-case behavior is in general not known. To guarantee that the WCET is measured, the program must be executed with *all* possible input values which is not feasible in practice. Second, the measurement often requires an instrumentation of the code, e.g., augmentation with instructions to control hardware timers. However, the certification of safety-critical systems often specifies that the validation is carried out on exactly the same code that is utilized in the final product. For example, in the avionic domain it is mandatory to use the same code for validation that is also later used in the airplane (*Test what you fly and fly what you test* [The04]). Hence, the measurement-based approach is not applicable in such environments.

Measurement-based approaches are currently the most common techniques found in industry since the hardware under analysis or the respective simulators are usually available. However, as there is no guarantee that the maximal program run time was measured, a safety margin is added to the measured execution times which often leads to highly overestimated timing results [SEG+06].

2.2.2 Static Approach

The static approach emphasizes the *safety* aspect and produces bounds on the execution time which are guaranteed to be never exceeded by the execution time of the program under analysis. Unlike the measurement-based approach, the static approach considers all possible input values to the task. To scale down the complexity of an exhaustive analysis of all values, the large number of possible input data is reduced using a safe abstraction. In addition, the static approach does not execute the code on real hardware or a simulator but analyzes the set of possible control flow paths through the program. Using abstract models of the hardware architecture, the path with the maximal execution time can be determined. This longest path is called the worst-case execution path (WCEP) and its length corresponds to the program's WCET. The success of the static approach highly depends on the abstract hardware models. If they are correct and specify the underlying system precisely, safe upper

bounds on the execution time of a program can be defined. However, it is in general hard to verify the correctness of the abstract models. Another drawback of the static analysis is that it might produce overestimated results if conservative decisions due to a lack of information during the analysis have to be taken. Furthermore, complexity of the static approach might be an issue if large programs are analyzed, leading to high analysis run times.

Currently, the static approach can be mainly found in academia. However, industry is recognizing the needs for safe WCET estimations. The growing industrial interest led to first applications of static WCET analysis in the automotive [BEGL05, SEG+06] and avionics [HLS00, FHL+01, SLPH+05] domain.

2.2.3 Hybrid Approach

The idea behind hybrid approaches is to combine concepts from the measurement-based and static approach. The hybrid approach identifies so-called *single feasible paths* (*SFP*) which are program paths consisting of a sequence of basic blocks where the execution is invariant to input data. To find SFPs at source code level, symbolic analysis on abstract syntax trees can be used [Wol02]. In the next step, the execution time of the SFPs is measured on real hardware or by cycle-accurate simulators. For input-dependent branches, input data for a complete branch coverage must be supplied. The execution time of these parts is also determined by measurements. In order to cover potential underestimation during the measurement, an additional safety margin is added to the measured execution time. Finally, the information of the SFPs is combined with techniques from the static approach to determine the longest path. The advantage of the hybrid approach is that it does not rely on complex abstract models of the hardware architecture. However, the uncertainty of covering the worst-case behavior by the measurement remains since a safe initial state and worst-case input can not be assumed in all cases. Moreover, instrumented code is required which may not be allowed in particular certification scenarios. This approach is used in the analysis tool suite *SymTA/S* [Sym10].

2.3 Basic Concepts for Static WCET Analysis

As shown in the previous section, the static approach is the only method to compute safe upper bounds on the execution time of a program. For compiler-based optimizations aiming at an automatic WCET minimization, reliable worst-case timing information is compulsory to achieve a systematic WCET reduction. Otherwise, unreliable timing information may result in adverse optimization decisions. This is the main reason why the WCET data exploited by the developed optimizations is computed statically.

Using the static approach for compiler optimizations is further motivated by practical reasons. First, many WCET-aware compiler optimizations operate iteratively,

requiring multiple updates of the worst-case timing information. The measurement-based approach does not provide the desired flexibility for a frequent WCET estimation. Second, both the static WCET estimation and compiler optimizations operate on similar code and data representations enabling a tight exchange of information between both approaches.

In the following, basic terms found in the context of static WCET analysis are introduced. Moreover, specific problems and requirements for this class of timing analysis are discussed.

2.3.1 Control Flow

A static WCET analysis requires information on the hardware timing such as the execution time of individual instructions. Thus, the analysis must be performed on the assembly or machine level of the code. In addition, to statically derive a timing bound of the program, its possible execution flows must be known. This information is provided by a control flow analysis which operates on assembly *basic blocks* [Muc97]:

Definition 2.1 (Basic block) A *basic block* is a maximal sequence of instructions that can be entered only at the first instruction and exited only from the last instruction.

As a consequence, the first instruction of a basic block may be the entry point of a function, a jump target, or an instruction that is immediately executed after a jump or return instruction. Call instructions represent a special case. Typically, they are not considered as a branch since the control flow continues at the immediate successor of the call instruction after the callee was processed. Thus, basic blocks may span across calls. However, if it can not be guaranteed that the control flow continues immediately after the call instruction, the call must be considered a basic block boundary. Examples are calls in the Fortran language with alternate returns or ANSI C exception-handling mechanisms, like `setjmp()/longjmp()`, which force the control flow to continue at a point in the program other than the call's immediate successor. Since the original source code language might be unknown at assembly level, timing analyzers must be conservative, thus handling call instructions as basic block boundaries.

Basic blocks can also be defined at other abstraction levels of the code. For example, high-level basic blocks defined at source code level represent sequential code consisting of statements.

A common data structure to represent the program's control flow is given by the *control flow graph* (*CFG*):

Definition 2.2 (Control flow graph) A *control flow graph* is a directed graph $G = (V, E, i)$, where nodes V correspond to basic blocks and edges $E \subseteq V \times V$ connect

two nodes $v_i, v_j \in V$ iff v_j is executed immediately after v_i. $i \in V$ represents the start node, called *source*, which has no incoming edges: $\nexists v \in V : (v, i) \in E$.

In literature, a CFG typically represents the control flow within a single function. On the contrary, an *interprocedural control flow graph* (*ICFG*) [LR91] models the control flow of an entire program across function boundaries. It constitutes the union of control flow graphs G_f for each function f augmented by call, return, entry, and exit nodes. Call nodes are connected to the entry nodes of functions they invoke, while exit nodes are connected to return nodes corresponding to these calls.

A possible execution path π through the ICFG is defined as follows:

Definition 2.3 (Path) A *path* π through the control flow graph $G = (V, E, i)$ is a sequence of basic blocks $(v_1, \ldots, v_n) \in V^*$, with $v_1 = i$ and $\forall j \in 1, \ldots, n-1 : (v_j, v_{j+1}) \in E$.

Using the definition of a path, the term *program* is defined as follows:

Definition 2.4 (Program) All possible paths through the control flow graph $G = (V, E, i)$ starting at i and ending in a sink constitute a *program* $\mathcal{P}$. A sink represents a block $s \in V$ with $\nexists v \in V : (s, v) \in E$.

The program path with the maximal length, which is expressed by the number of execution cycles when this path is executed, is referred to as the worst-case execution path (WCEP).

2.3.2 Processor Behavioral Analysis

The task of the *processor behavioral analysis* is the determination of timing semantics of the hardware components that influence the execution time of the program under analysis. In particular, processor components such as different memories (caches, scratchpads, etc.), pipelines, and branch predictions must be taken into account. The result of this analysis is an upper bound for the execution time of each instruction or basic block.

The processor behavioral analysis relies on an abstract model of the target architecture. For a sound approximation, it is not mandatory to model all of the processor's functionalities precisely. Simplified models may be sufficient if they can guarantee that timing semantics are handled in a conservative way, i.e., the worst-case timing is never underestimated.

The static WCET analysis of simple processors without caches and pipelines allows a separate consideration of instructions. To estimate the WCET of the program, the WCET of each basic block is computed by accumulating the execution times of its instructions [Sha89]. The consideration of instructions independent of their *execution history* is no longer valid for modern processors and may result in a WCET estimation which is not safe. The reasons are context-dependent execution times of instructions and timing anomalies.

2.3.2.1 Context-Dependent Timing Analysis

For modern processors, the execution of instructions depends on its *context*, i.e., which instructions were previously executed. Such situations may arise when a basic block b with more than one predecessor is executed. Depending on the previously executed block, the cache or pipeline behavior during the execution of b may vary, leading to different WCETs. Another example are loops. The first loop execution may have a higher execution time than the following executions since data must be initially loaded into the cache, leading to *compulsory cache misses*. Thus, a sophisticated WCET analyzer must exploit the knowledge of the execution history to estimate context-dependent upper timing bounds for instructions.

2.3.2.2 Timing Anomalies

Due to the complexity of hardware and software, timing analysis often only becomes feasible when abstraction is introduced. Abstraction allows the handling of unknown input data and enormous spaces of processor states but comes at the cost of less precision. Unknown parts of the execution states lead to non-determinism if decisions during the analysis rely on this unknown information [RWT+06]. The separate consideration of an instruction may lead to different execution times depending on different assumptions about the missing information. Intuitively, it would be assumed that a locally faster execution entails a decrease of the overall program's WCET. However, on modern processors this does not need to be true. A *timing anomaly* is defined as follows [Lun02, The04]:

Definition 2.5 (Timing anomaly) If the change ΔT_i of the overall execution time of an instruction i results in a change $\Delta T_{\mathcal{P}}$ of the global execution time of program $\mathcal{P}$, a *timing anomaly* occurs if either

- $\Delta T_i < 0 \rightarrow \Delta T_{\mathcal{P}} < \Delta T_i \vee \Delta T_{\mathcal{P}} > 0$, i.e., the acceleration of i leads to a smaller acceleration of the program's execution time or the program runs longer than before,
- $\Delta T_i > 0 \rightarrow \Delta T_{\mathcal{P}} > \Delta T_i \vee \Delta T_{\mathcal{P}} < 0$, i.e., the program's execution time is more extended than the delay of instruction i or the program execution is accelerated.

The crucial case for a WCET estimation is $\Delta T_i < 0 \rightarrow \Delta T_{\mathcal{P}} > 0$. It points out that a consideration of a local worst-case scenario at instruction i is not sufficient for a safe estimation of the WCET. Since verifying the absence of timing anomalies is provably hard, timing analyzers are forced to consider all possible scenarios, i.e., to follow execution through several successor states whenever a state with a non-deterministic choice between successor states is detected. This may lead to a *state explosion*. A first approach towards a more efficient WCET analysis, which exploits precomputed information about the abstract model of the hardware in order to safely discard analysis states, was presented in [RS09].

For modern processors, timing anomalies due to three different processor features were observed. *Speculation timing anomalies* arise when costs of an expensive branch mis-prediction exceed the costs of a cache miss [HLT+03]. *Scheduling timing anomalies* occur when dependent instructions are differently scheduled on the hardware resources, e.g., pipelines, leading to varying execution times of instruction sequences [Gra69, Lun02]. Therefore, scheduling anomalies can be observed on *out-of-order architectures*. But even *in-order architectures* may show timing anomalies. For example, on the Motorola ColdFire 5307 processor with its pseudo-round robin cache replacement strategy, an empty cache may not constitute the worst possible cache behavior w.r.t. the execution time [The04].

2.3.3 Flow Facts

The static WCET estimation is based on static program analyses which attempt to determine dynamic properties of the program under analysis without actually executing it [CC77]. Some of these properties are not decidable in general. In particular, the determination of the longest path requires the knowledge of loop iteration counts. However, a static loop analysis is generally not decidable for a concrete program semantics since it includes the proof of termination. Thus, an automatic loop analysis is only feasible for a restricted set of programs [EE00]. To enable an automatic timing analysis of arbitrary programs, the user is typically forced to support the WCET analyzer with program annotations about the loop iteration counts. In a similar way, recursion depths and target addresses of dynamic calls and jumps, which can not be statically determined, make a user annotation mandatory. If the program execution is dependent on input data, e.g., in sorting algorithms, the user annotations must respect all potential values that may be provided to the program. In literature, information about the dynamic behavior of a program is referred to as *flow facts* [Kir03].

Definition 2.6 (Flow facts) *Flow facts* give hints about possible paths through the control flow graph of a program $\mathcal{P}$. Flow facts can be expressed implicitly by the program structure or semantics as well as by additional information provided by user annotations.

2.3.4 Bound Calculation

In literature, three popular approaches to compute upper bounds on the execution time of a program $\mathcal{P}$ based on its (interprocedural) control flow graph are presented. All these approaches require additional control flow information about loop iteration counts and recursion depths which can be either provided by an automatic analysis or manually annotated flow facts.

The *path-based* approach [SEE01] models each possible path in the ICFG explicitly. For each of these paths, the path length is computed. The length of the longest path reflects the WCET. The main drawback of this approach is its complexity since the number of possible paths grows exponentially with the number of control flow branches. Therefore, heuristic search methods are possibly required. The *tree-based* approach [Sha89, CB02] traverses the abstract syntax tree of a program at source code level in a bottom-up manner and computes upper timing bounds for connected code constructs. The computation of the timing bounds is steered by predefined combination rules which state how the execution times of parent constructs are derived based on the execution times of their child constructs. After the computation, the current construct is collapsed and its WCET is propagated to its parent constructs. This approach makes an integration of flow facts difficult and relies on a clear mapping between the source and machine code which is not always given for optimized code.

Another approach, which is nowadays widely used, is the *implicit path enumeration technique* (*IPET*) [LMW95]. In this approach, the ICFG of a program is translated into a system of linear constraints. According to Kirchhoff's rules, the following constraints are generated:

$$\forall v, v' \in V, \quad \{(v, v') \in E\} : c(v) = \sum_{(v,v') \in E} trav(v, v') \wedge$$

$$\forall v, v' \in V, \quad \{(v', v) \in E\} : c(v) = \sum_{(v',v) \in E} trav(v', v)$$

$c(v)$ represents the execution count of block v and $trav(v, v')$ reflects the edge traversal count of edge $e = (v, v')$. Using these constraints, the control flow can be modeled as an *integer linear program* (*ILP*) [Chv83]. The equations warrant that the traversal counts of all incoming edges of v are equal to the traversal counts of all outgoing edges of v.

The problem to be solved is to maximize the overall flow through the ICFG. For this purpose, an objective function expressing the estimated WCET $WCET_{est}$ of the program $\mathcal{P}$ is formulated:

$$WCET_{est} = \sum_{v \in V} t(v) \cdot c(v)$$

$t(v)$ represents the WCET of basic block v computed by the processor behavioral analysis. This sum is a linear combination since the execution times of the blocks are constants in the integer linear program. Using the sum as objective function to the ILP as a maximization problem yields the estimated WCET of $\mathcal{P}$, while the results for $c(v)$ indicate how often v will be executed on the WCEP. For a more precise analysis, the model can be extended by execution contexts.

IPET is a flexible approach since it allows an easy addition of complementary constraints about the control flow, such as information about mutually exclusive paths that increase the precision of the upper bounds. Unlike the path-based approach, IPET does not explicitly model paths, thus the order of blocks lying on the

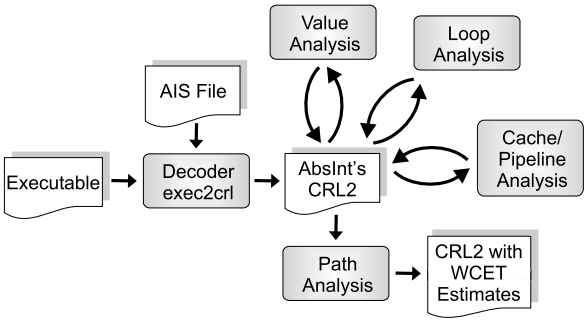

Fig. 2.1 Workflow of static WCET analyzer aiT

WCEP is not apparent. Despite the fact that solving of integer linear programs is $\mathcal{NP}$-complete, many large ILP problems can be solved in practice with a moderate amount of effort. For example, the WCET analysis provided by the tool aiT performed on an Intel Xeon 2.4 GHz machine takes 34 seconds for a simple cyclic redundancy check (CRC) computation [MWRG10] or 181 seconds for the encryption of a 32 bit message employing the commonly used secure hash algorithm (SHA) [GRE+01].

2.4 Static WCET Analyzer aiT

A leading static WCET analyzer is the tool aiT that is developed by the company AbsInt [Abs10]. Since it is used in this work for the calculation of upper timing bounds, its workflow will be briefly discussed in the following.

The modular architecture of aiT is depicted in Fig. 2.1 and consists of the following steps:

- As input, the timing analyzer expects a binary executable. It is disassembled by the tool *exec2crl* in order to construct an interprocedural control flow graph using the human-readable intermediate format *CRL2* (*Control Flow Representation Language*) [The02]. Moreover, flow facts from the parameter file called *AIS* might be imported to annotate the constructed ICFG with additional information. The initial CRL2 file is passed to the subsequent stages of aiT which annotate the file with further analysis results.
- The *value analysis* is based on an interval analysis to determine a sound approximation of potential values in the processor registers occurring at any possible program point. The values of the registers are exploited for the computation of possible addresses used for memory accesses. Moreover, the register values are exploited for the detection of *infeasible paths*, i.e., mutually exclusive paths that are never executed simultaneously in the same context.
- The static *loop analysis* tries to automatically determine the number of loop iterations for each loop in the ICFG. The analysis is based on pattern matching and involves results from the value analysis. aiT's loop analyzer succeeds for simple loops. Hence, user annotations for the remaining loops are required.

- The integrated *cache/pipeline analysis* relies on the register value ranges from the value analysis. The cache analysis tries to statically precompute cache contents. It performs a *must/may analysis* to classify memory references into sure cache hits and potential cache misses [FW99, HLT+03]. The pipeline analysis uses an abstract model of the target architecture to simulate the behavior of the pipeline [The04]. Considering timing anomalies and integrating results from the cache analysis, the pipeline analysis derives upper bounds on the execution time for each basic block depending on its context.
- The *path analysis* computes upper bounds for the overall program execution time. The computation is based on the IPET approach which exploits an ILP using the ICFG coded in CRL2 as well as timing results from the cache/pipeline analysis.

http://www.springer.com/978-90-481-9928-0

Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems

Lokuciejewski, P.; Marwedel, P.

2011, XVIII, 262 p., Hardcover
ISBN: 978-90-481-9928-0