Classification of WCET Analysis Techniques *

Raimund Kirner, Peter Puschner
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1, 1040 Wien, Austria
{raimund,peter}@vmars.tuwien.ac.at

Abstract

Worst-case execution time (WCET) analysis has become an active research area over the last decade. Various techniques have been developed to improve the WCET calculation methods for numerous features of the hardware. In parallel, attention has been paid to integrate the analysis techniques into modern software engineering processes.

In this paper we give an overview about the different aspects of WCET analysis. We clarify terms and categorise features of WCET analysis tools. Therefore we present a generic framework for WCET analysis and describe its fundamental operations. We present a classification scheme to test the applicability of WCET analysis tools for certain analysis requirements.

Keywords: Worst-Case Execution Time Analysis, Execution Times, Classification, Generic Framework

1 Introduction

Worst-Case Execution Time (WCET) analysis is about calculating an upper bound of a program's execution time. The WCET bound depends on the input data space of the program, the logic of the program code and the timing properties of the target hardware. In the last decade, extensive research has developed methods and techniques for static WCET analysis [28].

In general, static WCET analysis methods should provide safe and tight results. To avoid considering infeasible execution paths, several path descriptions and analysis methods have been developed. Their usability depend on their level of automation and the tightness of the results.

The increasing use of more advanced processor hardware has raised the need for more complex hardware modeling techniques. For example, to express and model timing effects of caches or pipelines, extensions of other approaches are needed.

There has been already a shift on the state of the art of writing programs for embedded computing from coding directly in assembly code to the use of programming languages like C. On top of the programming language, there also exist modeling environments with facilities for automatic code generation [13, 20]. The programming interface has an influence on the analyzability of programs. For example, high-level programming languages enforce a more restricted programming style than assembly languages and therefore may contain more precise implicit information about feasible paths.

Currently, there exist plenty of different WCET analysis approaches; each focusing on different hardware mechanisms of the target processor and providing different interfaces to specify the control flow behavior by code annotations.

This paper presents a classification of the features of WCET analysis tools in Section 2. It introduces terms for several analysis steps to set an end to the existing ambiguities between often used terms. In Section 3 it shows the fundamental components of each WCET analysis framework and analyses their functions and properties.

Based on the observed classifications for WCET analysis tools, a scheme is shown to evaluate WCET analysis tools for applicability under certain requirements (Section 4). The application of this scheme is demonstrated in Section 5.

2 Aspects of WCET Analysis

The goal of worst-case execution time (WCET) analysis is to calculate the maximum execution time of a given piece of code for a specific run-time environment (e.g., specific hardware architecture).

To classify WCET analysis techniques, it is necessary to

*This work has been partially supported by the FIT-IT research project "Model-Based Development of distributed Embedded Control Systems (MoDECS)" and the ARTIST2 Network of Excellence of IST FP6.

define clear and intuitive terms. For example, the unfortunate term *low-level analysis* is used with different meanings in the literature:

- One of its meanings is *exec-time modeling*, i.e., to assign execution cycles to all statements of assembly programs before calculating the final WCET value.

- In a different interpretation, the extraction of control flow information and path analysis are subsumed as *high-level analysis*. *Low-level analysis* is used to describe all aspects that are not part of the *high-level analysis*. This interpretation of *low-level analysis* covers more functionality than the first.

- Another meaning of *low-level analysis* comes from *representation levels*, where for a source program to be analyzed the WCET analysis tool operates on a different representation level of the program.

Calculating the WCET is a quite complex problem. There are different orthogonal aspects that have to be considered. Some are related to the input format of the program to be analyzed (e.g.: assembly code) or the target machine for which the WCET is of interest.

Another point is the WCET analysis method used by the tool. In general, the WCET analysis method cannot be transparent to the user, since it is not possible to calculate the WCET automatically for an arbitrary program. This is due to the fact that the WCET calculation problem can be transformed to the well-known *Halting Problem* [24], which is provably not solveable in the generic case. To overcome this limitation, two techniques can be used:

- Restricting the programming language, so that only analyzable code can result. This approach limits the usability of the language to domain-specific solutions.

- Adding additional knowledge about the possible program control flow that cannot be derived automatically from the program code. This approach supports the use of generic programming languages but requires the user to bring in domain-specific knowledge (e.g., loop bounds, ranges for instantiation of input parameters).

To describe properties of programs to be analyzed, the following definitions will be used throughout this work:

Definition 1 Control flow paths $CFP(\mathcal{P})$ *describe the set of possible execution paths of program $\mathcal{P}$ with applied execution constraints. Such constraints are for example ranges for the value instantiation of input parameters. $CFP(\mathcal{P})$ is a description of the set of different execution traces of program $\mathcal{P}$. In case a program has unbounded loops this set is unbounded. We distinguish the following two types of CFP:*

- $CFP_{opt}(\mathcal{P})$ *... Control flow paths for a program $\mathcal{P}$ as seen by the omniscient observer. $CFP_{opt}(\mathcal{P})$ is an abstract and exact description of all possible program execution traces. $CFP_{opt}(\mathcal{P})$ includes only these execution traces that can really occur on program execution. One can reduce $CFP_{opt}(\mathcal{P})$ by specialization of the program execution (e.g., by assuming input parameters of restricted shape or value range).*

- $CFP_{ff}(\mathcal{P})$ *... Control flow paths of a program $\mathcal{P}$ described by flow facts ff. $CFP_{ff}(\mathcal{P})$ is an approximation of $CFP_{opt}(\mathcal{P})$ by considering a set of flow facts ff: $CFP_{opt}(\mathcal{P}) \subseteq CFP_{ff}(\mathcal{P})$. $CFP_{ff}(\mathcal{P})$ can be described as the by ff spawn closure of execution paths for a program $\mathcal{P}$.*

Definition 2 *$CFP_{WCET,opt}$ is an execution trace that yields the optimal solution of the WCET calculation. $CFP_{WCET,opt}$ is derived from CFP_{opt} (analogously, $CFP_{WCET,ff}$ is an execution trace for the calculated WCET, depending on ff).*

Definition 3 Flow facts *ff give hints about the possible CFP of a program. The CFP over ff is called CFP_{ff}. Flow facts can be expressed implicitely by the structure of the program itself as also by additional user information.*

Definition 4 Implicit flow facts *ff_{impl} are ff that are given implicitely by the program structure and semantic. If the CFP_{opt} of a program does not depend on input variables or external events, it can be completely described by ff_{impl}.*

A concrete WCET analysis tool may be only capable to extract a subset of FFI from the program code. We call this set $ff_{impl,tool}$.

Definition 5 Flow facts by annotations *ff_a are ff that are given explicitely by code annotations. ff_a are used to simplify WCET tool implementation (avoiding complicated code analysis) or to bring in additional information to make WCET analysis feasible and tight.*

The features of WCET analysis frameworks can be divided into three orthogonal aspects as shown in Fig. 1. They are called *representation level*, *exec-time modeling* and *flow facts*. Along each axis typical examples are shown. The ordering along the axes show some kind of complexity level (it increases from the center), but this is only to give an idea of the magnitude and not to compare certain components.

The meanings of these three aspects are as follows:

2.1 Representation Level

The coding of a program and the WCET analysis may be performed at different *representation levels*. To obtain

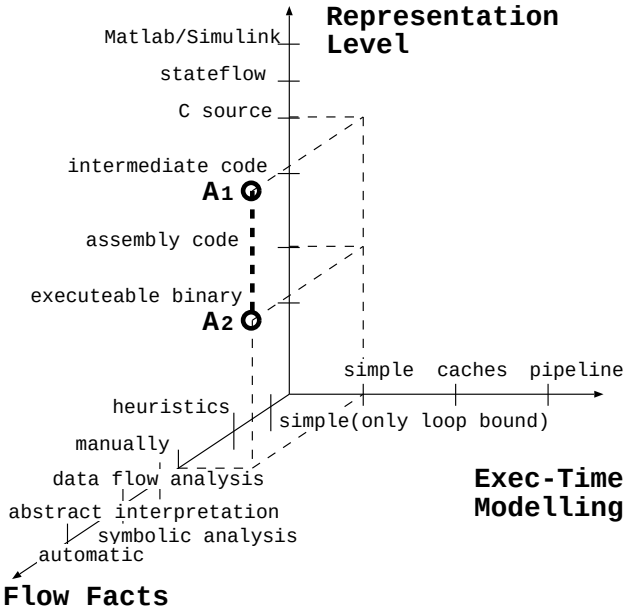

Figure 1. Orthogonal Aspects of WCET Analysis

accurate and tight time bounds, WCET analysis typically operates at assembly/object code level.

Programming in assembly code should only be done where it is strictly necessary, e.g., due to resource limitations. Typical representations for program development are in 3GL[1]. Actually, there has been done research on WCET interfaces for 3GL like Euclid [21], Modula2 [34], Java [6], C [19, 25], etc. Furthermore, tools to model an application by its algorithm are for example MATLAB/Simulink[2] or the Statemate Statechart system [16].

As discussed above, it is required to have some flow information (ff) about the possible CFP. When the representation level of the programming language differs from that where the WCET analysis is done, compilers have to transform the ff to the level of analysis (we use the term compiler here for all kind of program transformations). Compilers typically provide powerful optimizations that change the structure of the program down to assembly level dramatically.

This leads to the challenge to transform the ff from the programming language down to assembly level. Therefore, methods are required to keep the ff useful even in the presence of optimizing code transformations.

Fig. 1 shows an example, where the program is coded in C (marked as A1) and the analysis is done at assembly level (marked as A2).

[1] 3GL... Third Generation Programming Language

[2] http://www.mathworks.com/

2.2 Flow Facts

Flow facts ff are hints that describe constraints on the possible CFP of a program. Possible sources for ff are syntactic and semantic information of the program code or additional annotations (ff_a). ff_a can be given as annotations inside the source code , on files that are separate from the source code, or interactively.

Methods for characterizing flow facts can have different level of automatism. In the optimal way ff are extracted automatically from the programs syntax and semantic. Semantic analysis for example can use data and control dependence analysis [1] of the program, abstract interpretation [10] as well as symbolic evaluation [7]. Such methods tend to be very complex when best quality of ff is required.

In cases where $ff_{impl,tool}$ are not sufficient to calculate a WCET bound or achieve the desired quality in the WCET calculation, additional flow facts are needed, that are given manually (ff_a). At least, ff that are input-data dependent, require additional ff_a to be determined.

For a CFP that contains cycles (loops), the knowledge of the maximum execution number of the backward edges (loop bounds) is mandatory for WCET analysis. To obtain tighter time bounds, further knowledge about infeasible paths has to be brought in.

The following is a list of examples for different levels of detail of execution frequency constraints (some kind of ff_a):

- loop bounds (exact versus safe, upper bounds)
- loop bounds + some additional constraints (e.g., loop sequences [27], maximum execution count of a certain operation within a specific scope)
- arbitrary constraints on execution count of blocks
- a single set of conjunctive constraints (versus disjunctive sets of conjunctive constraints)
- one set of constraints (only conjunctive) versus also disjunctive constraints (necessary for completeness of constraints that model the algorithmic program behavior)

Another type of ff_a are constraints that describe the execution order of operations. An example for this can be found in [12] where the authors describe execution frequency constraints that also allow to address certain iteration ranges within a loop.

Other types of ff_a are constraints on the ranges of value instantiations of input parameters [15]. Transforming such ff_a require advanced analysis techniques to transform them so that they can be used by the underlying WCET calculation method.

For domain-specific solutions the calculation of the CFP can be simplified by using a programming language with a very restrictive syntax.

2.3 Exec-Time Modeling

Exec-time modeling refers to assessing the timing properties of the runtime environment. It models the behavior of the constituents of the code (instructions/statements) on the target machine (real hardware, virtual machine, ...) in a way that is feasible and sufficient enough to get safe and tight WCET bounds. In a simple case, the execution time of each statement of the target hardware only depends on the type of its arguments.

The execution time of statements in modern CPUs that contain performance enhancing hardware features like pipelines, caches or branch prediction typically depend on the value of the arguments and the context inside the program.

Precise exec-time modeling tends to result in complex analysis algorithms and is often only feasible up to a certain limit. Therefore, the hardware characteristics are often modelled only partially or in a simplified manner.

To give some examples, exec-time modeling of caches has to deal with aspects like:

direct mapped caches,

set/fully associative caches,

instruction/jump caches,

data caches,

unified caches,

multiple cache hierarchies.

The effort required for *exec-time modeling* of pipelines depends strongly on the concrete pipeline concept. Engblom [11] has analyzed the following pipeline concepts for *exec-time modeling*:

simple scalar pipelines,

scalar pipelines,

VLIW[3] (statically scheduled pipeline),

superscalar in-order pipeline (dynamically sched.),

superscalar out-of-order pipeline.

Engblom used a pragmatic approach to perform exec-time modeling, that relies on a cycle-accurate trace-driven simulator. Proving the correctness of such a simulator experimentally is not feasible. Still, this approach is a useful best-effort method to overcome the conceptional limitations due to complexity in practice.

[3] VLIW... very long instruction word

Colin et al. [9] modelled the branch-prediction behavior of the Intel Pentium processor. They modelled the instruction cache, the branch prediction mechanism and the pipeline. The authors use the tool Salto[32] to construct the pipeline reservation table of each instruction.

Li et al. [22] modelled instruction caches (direct mapped as well as set associative) and data caches. They constructed a timing model by generating constraints, which represent some kind of flow facts. The usage of this approach is limited, because the resulting complexity of the *execution scenario calculation* becomes infeasibly complex for real-world programs. The *execution scenario calculation* is the final analysis step where the flow facts and the exec-time model are used to calculate the WCET bound.

2.4 Dependence of WCET Aspects

The three aspects of WCET analysis, as shown in the "feature-space" in Fig. 1, are not completely independent.

For example, different levels of detail in *exec-time modeling* require different sets of *ff*. To allow retargetable WCET analysis, the expressiveness of the *ff* must be designed powerful enough to cover all target hardware platforms of interest. Otherwise it is not possible to model all hardware features, even though the underlying method for *execution scenario calculation* the would support it.

After collecting all available *ff*, they are transformed into the format required by the *execution scenario calculation*. The format of the *ff* must be compatible with the calculation method. For example, calculating parametric results [5], i.e., a timing formulae instead of a constant numeric value, requires appropriate *ff*. The same is true for the *exec-time modeling*. Some approaches enforce strictly separated analysis passes, where the result of the *exec-time modeling* represents already resolved execution times for single statements/blocks. Other approaches use *exec-time modeling* to generate constraints that are considered in the *execution scenario calculation*.

3 The Process of WCET Analysis

In this section we introduce the fundamental components of a WCET analysis framework and analyze their functionality.

3.1 Generic WCET Analysis Framework

We now introduce a generic WCET analysis framework to show its main components and how they are used. In the following is shown, how existing WCET analysis frameworks fit to this abstraction. Existing frameworks may look simpler, because they use less complex and less powerful

methods. Other approaches only differ in the variety of features and functionality inside the main components.

Fig. 2 shows the components of the generic WCET analysis framework. The input is the source representation of the program. The representation level of the input program and the level where the exec-time modeling of the analysis has to be performed, together determine whether there is additional information necessary about the compilation process. For example, to maintain consistency of the flow facts in case the compiler performs optimizing code transformations it is required that the *compilation* and the *transformation of flow facts* work tightly coupled (as indicated by the dotted line).

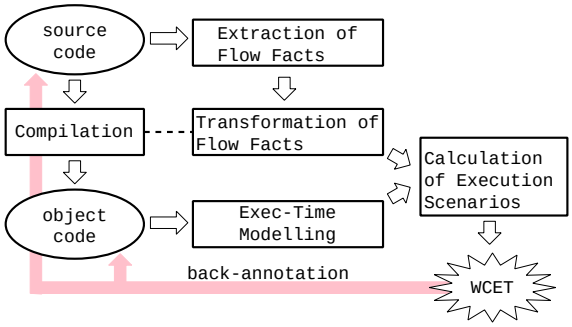

Figure 2. Generic WCET Analysis Framework

Some frameworks directly read the compiled object code as input and request the user interactively for the required flow information [22]. In this case the *extraction of flow facts* and *transformation of flow facts* is left to the intellectual power of the user.

3.2 Formal Definitions

We use the operator $\Diamond$ from modal logic to model the relation *"it can be"* ($\Diamond$ does *not* mean that the expression must be true under at least one interpretation or variable instantiation).

Definition 6 (Intermediate Representations) *The WCET analysis can be divided into several phases. The following operations and intermediate results are considered:*

- src *... source representation of the program*
- $obj = c(src)$ *... object code of the program*
- $ff = e(src)$ *... flow facts that give hints about the possible execution scenarios of the program.*
- $ff_c = \tilde{c}(ff)$ *... transformed flow facts (from source to object code, including symbolic or numeric calculations)*
- $m_t = t_M(obj)$ *... concrete hardware timing model.*
- $WCET_{calc} = \omega(ff_c, m_t)$ *... calculated WCET*
- $SC_\omega = \beta_s(WCET_{calc}, src)$ *... WCET, back-annotated to source level*
- $OC_\omega = \beta_o(WCET_{calc}, obj)$ *... WCET, back-annotated to object level*

3.3 Extraction of Flow Facts

Calculating the WCET by only using ff_{impl} is in general impossible. Therefore, the use of additional ff_a is required:

$$ff = ff_{impl} \cup ff_a \tag{1}$$

As shown in Equ. 2, redundant ff_a are often used to simplify the extraction of ff. The drawback is that specifying ff_a explicitely could be a source for errors if it is done manually by the user.

$$\Diamond(ff_{impl} \cap ff_a \neq \{\}) \tag{2}$$

Definition 7 *$C_P(ff)$ is the by ff generated closure of execution paths. "By ff generated" means that the set of all the execution paths possible from the syntactic structure of the program are taken, but constrained by the flow facts ff. This is the same set as described by CFP_{ff} (Def. 3): $CFP_{ff} = C_P(ff)$.*

3.3.1 The Dualism between ff and CFP_{ff}

To perform WCET analysis, the minimum required set of ff has to contain the syntactic structure and bounds for all loops. We call this minimal set of flow facts $ff_{syntax,lb}$. $ff_{syntax,lb}$ generates the maximal set of execution traces $CFP_{syntax,lb}$. The abstract flow facts that would be required to build CFP_{opt} (Def. 3) are called ff_{opt}. All WCET analysis frameworks support some kind of ff within these two extrema. It requires precise and flexible ff to minimize one potential cause for WCET overestimation - the infeasible paths: $C_P(ff) - CFP_{opt}$.

The CFP_{ff_x} for different ff_x are in partial order. Fig. 3 shows an example using a *Hasse Diagram*. From the structure of the partial order we can construct a lattice $L\langle M, \cap, \cup, \bot, \top\rangle$ where $M = \bigcup_{x \in X} CFP_{ff_x}$, $\bot = CFP_{opt}$ and $\top = CFP_{syntax,lb}$. This formalism shows intuitively the effect of enriching ff.

Assuming that all ff_x are in normalized form without redundancy, we can show the dualism between changes in CFP_{ff} and ff. This normalization is an abstract model as it is in general not trivial to define a normalized form for all kinds of ff because it would also depend on the underlying method for calculating the execution scenarios. From that we get the definition of the lattice $L_D\langle M_D, \cap, \cup, \bot, \top\rangle$ where $M_D = \bigcup_{x \in X} ff_x$, $\bot = ff_{syntax,lb}$ and $\top = ff_{opt}$.

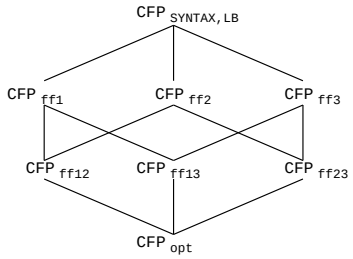

Figure 3. Partial order of CFP_{ff} and CFP_{opt}

$$ff_z = ff_x \cup ff_y \iff CFP_{ff\,z} = CFP_{ff\,x} \cap CFP_{ff\,y}$$
$$ff_z = ff_x \cap ff_y \iff CFP_{ff\,z} = CFP_{ff\,x} \cup CFP_{ff\,y} \quad (3)$$

The dualism of L_D and L in modifying ff and CFP_{ff} is shown in Equ. 3. It demonstrates how enriching ff will bring CFP_{ff} closer to CFP_{opt}. Additionally, we can derive the following rules from Fig. 3:

- If $(CFP_{ff\,x} = CFP_{opt})$, then ff_x is optimal.
- If $(CFP_{ff\,x} \cap CFP_{opt} \subset CFP_{opt})$, then ff_x is an invalid path description and can cause an underestimation of the WCET.

3.3.2 Methods

As already mentioned in Section 2.2, methods for characterization of ff can have different level of automatism. To bring the ff in a format useful for the WCET calculation method, $e(src)$ has to evaluate and convert ff. $e(src)$ provides implicit as well as explicit ff available at src-level: $ff_{impl} \cup ff_a = ff = e(src)$.

The syntactic structure can be extracted easily. The same is true for ff_a that act as simple structure information (e.g., loop bounds). ff_a that describe the program behavior indirectly, will be called *indirect* ff_a. An example for *indirect* ff_a are symbolic expressions that describe a loop bound in an algorithmic way similar to the program code.

3.4 Compilation

In order to analyze a program for its WCET it is necessary to transform it from its source representation (src) to the representation where the analysis is done (obj). This transformation $obj = c(src)$ (as defined in Def. 6) is in common the program compilation. This compilation is a surjective projection from src to obj. The projection is defined by the compiler version and the activated compiler switches. Therefore, it is not possible to match the control structures from obj directly with that from src for all kind of code optimizations of the compiler.

Obviously, for the case that src and obj are on the same representation level, no transformation is required. A typical example of this is writing and analyzing a program at assembly level. Another case is described in [26], where WCET analysis is done directly at the C language level (actually on a small subset of C).

3.5 Transformation of Flow Facts

Whenever a compiler transforms the program in such a way that the control flow is changed, it is required to transform the information about the control structure and other ff of the program in accordance with the program compilation. We have defined in Def. 6 the operation for transforming ff as $ff_c = \tilde{c}(ff)$.

$\tilde{c}(ff)$ must work in close connection with the compilation process $c(src)$. Using the debug information of the compiler to implement $\tilde{c}(ff)$ can sometimes work as a simple mapping solution. In case of strong code optimizations it is required to get additional support by the compiler.

The simplest approach would be to transform ff manually from src down to obj, as done by Li et al. [22] (the authors enforce the user of the tool to do this by interactively requesting flow facts like loop bounds from the user). This technique is simple to implement but for the user it becomes time-consuming and error-prone.

Another approach would be to extend the compiler to support the transformation of ff in case of code optimizations [19].

3.6 Exec-Time Modeling

To perform WCET analysis, it is required to derive a concrete time model m_t for the program obj (not necessarily the object code).

The operation to derive m_t is $t_M(obj)$ (Def. 6). As mentioned in Section 2.4, the semantic of m_t must be compatible with the method for execution scenario calculation.

The construction of an accurate m_t together with the search for a minimal CFP_{ff} are most challenging to minimize the overestimation of the WCET.

To model some hardware features accurately, it may be necessary to perform $t_M(obj)$ at inter-procedural level, especially for recursive or short callee functions.

3.7 Calculation of Execution Scenarios

As shown by Equ. 4, the calculation of $WCET_{calc}$ depends on a sequence of previous operations.

$$WCET_{calc} = \omega(\tilde{c}(e(src)), t_M(c(src))) \quad (4)$$

3.7.1 The Worst-Case Execution Trace

In Section 3.3 we have seen that all CFP_{ff_x} are in partial order and converge to CFP_{opt} by enriching ff_x (Fig. 3).

The $WCET_{calc}$ calculated by $\omega(ff_c, m_t)$ has a corresponding execution trace $CFP_{WCET,ff}$ which takes $WCET_{calc}$ to execute. There is a partial order between CFP_{ff} and $CFP_{WCET,ff}$ as shown in Equ. 5. The same is true for $CFP_{WCET,opt}$. It is also interesting to note, that if $(CFP_{WCET,opt} = CFP_{opt})$ then the program has a *single path* structure.

$$CFP_{WCET,opt} \subseteq CFP_{opt}$$
$$CFP_{WCET,ff} \subseteq CFP_{ff} \quad (5)$$

Comparing $CFP_{WCET,ff}$ and $CFP_{WCET,opt}$ we get some disappointing results. As shown in Fig. 4, the various CFP_{WCET,ff_x} do not converge into $CFP_{WCET,opt}$. For example, CFP_{ff_1} and $CFP_{ff_{12}}$ yield to at least partially different $CFP_{WCET,ff}$. Furthermore, $CFP_{ff_{23}}$ shows that we can get different $CFP_{WCET,ff}$ for the same ff. The reason comes from the fact, that *exec-time modeling* also influences $WCET_{calc}$ (Def. 6).

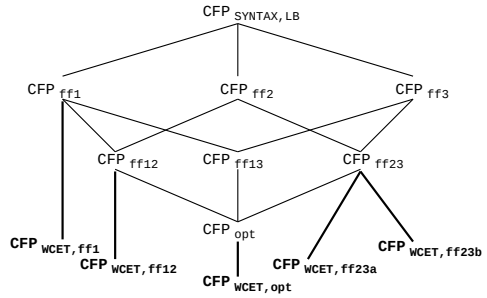

Figure 4. Partial order of CFP_{ff} and $CFP_{WCET,ff}$

The fact, that the calculated $CFP_{WCET,ff}$ can be different than the execution trace for the optimal WCET solution is formulated in Equ. 6. For the other case ($CFP_{WCET,ff} = CFP_{WCET,opt}$) we would have found the *optimal path solution*. But more important, Equ. 7 states that if we have not found the *optimal path solution*, we have found a $CFP_{WCET,ff}$, outside of in reality possible execution paths. Beside incomplete exec-time modeling this is a major reason for *overestimating* the WCET.

$$\Diamond(CFP_{WCET,ff} \neq CFP_{WCET,opt}) \quad (6)$$
$$(CFP_{WCET,ff} \neq CFP_{WCET,opt}) \longrightarrow (CFP_{WCET,ff} \notin CFP_{opt}) \quad (7)$$

Theorem 8 *It is* not safe *to extrapolate from a calculated $WCET_{calc}$ value to the effects of additional hardware properties. Applying a different timing model can result in a different $CFP_{WCET,ff}$.*

The main result of investigating $CFP_{WCET,ff}$ is given in Theorem 8. From this theorem is also follows that different *exec-time modeling* can yield different $CFP_{WCET,ff}$ for the same ff.

For example, it can be experienced that when adding DRAM refresh cycles to the system that the previously calculated $CFP_{WCET,ff}$ will change [4].

3.7.2 Calculation Methods

Typical calculation methods for $WCET_{calc} = \omega(ff_c, m_t)$ are integer linear programming (ILP) [30] (global algorithm), tree based calculations (timing schema) [8, 29] (hierarchical tree-based algorithm) or path-based calculations [17, 33]. Also hybrid solutions could be useful. The result of the *execution scenario calculation* is $WCET_{calc}$, but some approaches also deliver additional information like $CFP_{WCET,ff}$ or the execution frequency of each statement.

3.7.3 Back-Annotation of Results

To examine the timing behavior of a program and looking for best places to optimize code for a lower WCET, it is required to split the $WCET_{calc}$ to its contribution to blocks of certain granularity. It is desired to know the WCET contribution for each single statement.

Back-annotation can be done on several representation levels within *src* ($SC_\omega = \beta_s(WCET_{calc}, src)$) and *obj* ($OC_\omega = \beta_o(WCET_{calc}, obj)$). It depends on the method used for *execution scenario calculation*, how much information is available.

4 Classification Criteria for WCET Analysis Tools

In this section we present a scheme to evaluate WCET analysis tools for their useability on a certain target hardware with requirements on the calculation quality. Once each WCET analysis tool has been classified, anyone interested in using WCET analysis can simply compare the features required by him with the features provided by the tools.

4.1 Representation

We propose a graphical method, *kivat-diagrams*[4], to show the features and requirements of the target hardware and how a given WCET analysis framework does conform

[4] also known as *radar diagram*

to them. Therefore, we map the "feature-space" as shown in Fig. 1 into a two-dimensional representation.

In principle, all subcategories of features are drawn as extra beam in the kivat diagram. Each subcategory should represent an increasing level of quality. To achieve this, each of the three aspects in WCET analysis has to be subdivided, because the axis in Fig. 1 do not show a strict level of quality.

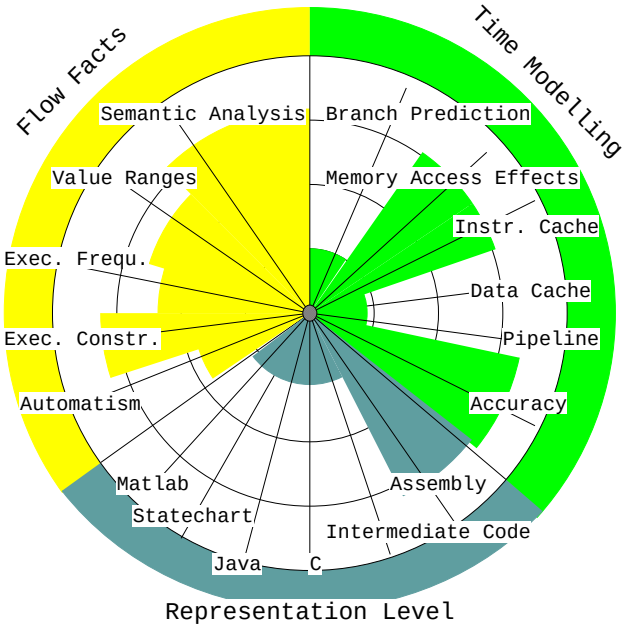

Figure 5. Kivat Diagram of WCET Analysis Features

An interesting point is, how to divide the features to have a strict order of quality on each feature beam in the diagram. We demonstrate this for the categorization of caches. For caches we differentiate between instruction and data caches, which can also interfere in form of unified caches. A possible classification for data and instruction caches is given in Table 1. To improve preciseness, criteria level F and G could also be modelled as separate criteria.

Quality	Data Cache	Instruction Cache
G	unified cache	unified cache
F	multi-level	multi-level
E	fully-associative	fully-associative
D	set-associative	set-associative
C	direct-mapped	direct-mapped
B	no cache	jump cache
A	no cache	no cache

Table 1. Quality Criteria for Data and Instruction Caches

This representation allows a tradeoff between simplicity and preciseness.

4.2 How to Classify a WCET Analysis Framework

After identifying all relevant features for a WCET analysis framework it is possible to construct a *kivat diagram* for all the features that are required for the current target hardware and the program source format. In the sector of *flow facts*, one can select the desired form of *ff* to enable WCET analysis. An example for such a specification on the required features of a WCET analysis tool is shown in Fig. 6a).

The capabilities of a concrete WCET analysis tool are drawn by black sectors as in Fig. 6b).

Now it can be tested whether the tool is applicative to the specification by overlapping both diagrams. The test for the tool of Fig. 6b) is shown in Fig. 6c). This test shows that there are still features of the specification that are not covered by the tool. Using the tool anyway would introduce additional pessimism in the analysis.

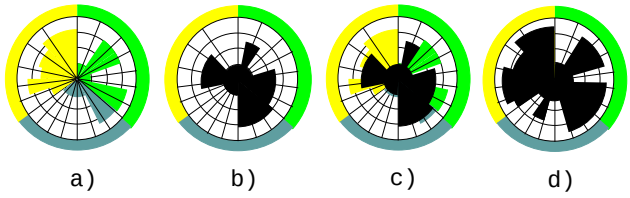

Figure 6. Matching of WCET Analysis Features

Fig. 6d) shows the test for a different WCET analysis tool. As can be seen by the full coverage, this tool is appropriate for the desired specification of a WCET analysis tool in Fig. 6a).

5 Case Study

After presenting these generic classification criteria for a WCET analysis framework, we present a case study with several WCET analysis frameworks. Of course, this cannot be a complete list, but it should be sufficient to demonstrate the classification according to the scheme in Fig. 5.

The features of the selected WCET analysis frameworks are summarized in Table 2. This table describes the features of each WCET aspect (*exec-time modeling*, *flow facts* and *representation level*) and the *path analysis* method used by the *calculation of execution scenarios*.

It is important to note, that the provided quality of certain features may not be at the same level for each approach. Therefore, it is also necessary to study the quality aspects of the features sufficiently. For example, pipeline analysis can

Authors (incomplete)	Kirner, Puschner	Healy,Ko, Mueller	Park,Shaw	Colin,Puaut	Puschner, Vrchoticky	Li,Malik, Wolfe	Ferdinand, Wilhelm
Bibliography	[3, 19, 20]	[2, 18, 35]	[25]	[9]	[31, 34]	[23]	[14]
Instr. Cache	jump-cache	set-assoc.	none	set-assoc.	none	set-assoc.	set-a./unified
Data Cache	none	direct-mapped	none	none	none	set-assoc.	set-a./unified
Branch Predict.	none	none	none	BTP (Pent.)	none	none	none
Pipeline	simple scalar	simple scalar	none	superscalar	none	none	superscalar
Mem. Access Loc.	different (3)	one	one	one	one	one	different (2)
DRAM Refresh	none	none	yes	none	none	none	none
Accuracy	safe	safe	safe	safe	safe	safe	safe
Automatism	syntax only	dafaflow anal., loop bounds	syntax only	symb. eval.	syntax only	syntax only	abstr. interpr.
Exec.Frequ. Constr.	yes	none	yes	none	yes	yes	n.a.
Infeas.Paths Constr.	yes	none	yes	none	yes	yes	n.a.
Value Ranges	none	yes	none	none	none	none	n.a.
Program Input	MATLAB, C	C, Ada	C	C	ModulaR	C	C
Analysis Performed	assembly	object	C	assembly	assembly	object	object
Spec. of Flow Facts	C src.	interactively	spec.file	C src.	ModulaR	interactively	C src.
Tool User Interface	graphically	graphically	cmd.line	cmd.line	cmd.line	graphically	cmd.line
Path Analysis	ILP	path-based	tree-based	tree-based	ILP	ILP	ILP

Table 2. Case Study on Existing WCET Analysis Frameworks

be done globally (tighter results) or locally at basic block level (more overestimation).

Using our classification criteria allows to evaluate and compare WCET analysis frameworks on a fine granularity. This eases the choice of the appropriate WCET analysis framework for certain requirements. For the purpose of summarizing the features of several tools we have used the representation of a single table. However, for the selection of a proper WCET analysis tool one would preferably first construct a kivat diagram showing all the required features for the analysis (Fig. 6a). Then, one identifies adequate WCET analysis tool by testing whether the kivat diagram of the tool covers all the required features.

6 Summary and Conclusion

This paper has given a classification of existing WCET analysis techniques. The focus has been on clarifying terms related to WCET analysis and categorising features of WCET analysis techniques.

Three fundamental orthogonal aspects of WCET analysis have been identified: *representation level*, *flow facts* and *exec-time modeling*. A "feature-space" has been constructed that assigns the different features of WCET analysis techniques to these three aspects.

Based on the different features and requirements of WCET analysis frameworks a classification scheme has been introduced. With this scheme it is easy to evaluate and identify WCET analysis techniques that are appropriate for the requirements of a certain target hardware and tool interface.

A case study illustrated the use of our classification scheme for a number of existing WCET analysis frameworks.

References

[1] A. W. Appel. *Modern Compiler Implementation in C*. Press Syndicate of the University of Cambridge, 1999. ISBN 0-521-58390-X.

[2] R. D. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding Worst-Case Instruction Cache Performance. In *Proc. 15th Real-Time Systems Symposium (RTSS)*, pages 172–181, Brookline, Massachusetts, Dec. 1994.

[3] P. Atanassov, R. Kirner, and P. Puschner. Using real hardware to create an accurate timing model for execution-time analysis. In *International Workshop on Real-Time Embedded Systems RTES (in conjunction with 22nd IEEE RTSS 2001)*, London, UK, Dec. 2001.

[4] P. Atanassov and P. Puschner. Impact of dram refresh on the execution time of real-time tasks. In *International Workshop on Application of Reliable Computing and Communication (WARCC)*, Dec. 2001.

[5] G. Bernat and A. Burns. An approach to Symbolic Worst-Case Execution Time Analysis. In *Proc. 25th Workshop on Real-Time programming*, Palma, Spain, May 2000.

[6] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis using Java Byte Code. In *Proc. 6th International EUROMICRO conference on Real-Time Systems*, Stockholm, June 2000.

[7] J. Blieberger. Data-flow frameworks for worst-case execution time analysis. *Real-Time Systems*, 22:183–227, 2002.

[8] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *RTS*, 18(2):249–274, May 2000.

[9] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proc. 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, Netherland, June 2001. Technical University of Delft.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[11] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Acta Universitatis Upsaliensis, Uppsala, Sweden, Apr. 2002.

[12] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000.

[13] E. Erpenbach and P. Altenbernd. Worst-Case Execution Times and Schedulability Analysis of Statecharts Models. In *Proc. 11th Euromicro Conference on Real Time Systems*, York, June 1999.

[14] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. of the 1st International Workshop on Embedded Software (EMSOFT 2001)*, pages 469–485, Tahoe City, CA, USA, Oct. 2001.

[15] J. Gustafsson. *Analysing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Uppsala University, Uppsala, Sweden, May 2000.

[16] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4), Oct. 1996.

[17] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1), Jan. 1999.

[18] C. A. Healy, M. Sjödin, and D. B. Whalley. Bounding Loop Iterations for Timing Analysis. In *Proc. IEEE Real-Time Technology and Aplications Symposium*, pages 12–21, June 1998.

[19] R. Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, Vienna, Austria, May 2003.

[20] R. Kirner, R. Lang, G. Freiberger, and P. Puschner. Fully automatic worst-case execution time analysis for Matlab/Simulink models. In *Proc. 14th Euromicro Conference on Real-Time Systems*, pages 31–40, Vienna, Austria, June 2002. Vienna University of Technology, IEEE.

[21] E. Klingerman and A. D. Stoyenko. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, 12(9):941–989, Sep. 1986.

[22] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proc. IEEE Real-Time Systems Symposium*, pages 298–307, Dec. 1995.

[23] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proc. 17th Real-Time Systems Symposium*, pages 254–263. IEEE, Dec. 1996.

[24] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

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

[26] C. Y. Park and A. C. Shaw. Experiments with a Program Timing Tool based on a Source-Level Timing Schema. *Computer*, 24(5):48–57, May 1991.

[27] P. Puschner. Ermittlung der maximalen Abarbeitungszeit von Programmen. Master's thesis, Technische Universität Wien, Vienna, Sep. 1988.

[28] P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

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

[30] P. Puschner and A. V. Schedl. A Tool for the Computation of the Worst Case Task Execution Times. In *Proc. Euromicro Workshop on Real-Time Systems*, pages 224–229, Oulu, Finland, June 1993.

[31] P. Puschner and A. V. Schedl. Computing Maximum Task Execution Times – A Graph-Based Approach. *The Journal of Real-Time Systems*, 13:67–91, 1997.

[32] E. Rohou, F. Bodin, and A. Seznec. Salto: System for assembly-language transformation and optimization. In *Proc. 6th Workshop on Compilers for Parallel Computers*, Dec. 1996.

[33] 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.

[34] A. Vrchoticky. Compilation Support for Fine-Grained Execution Time Analysis. In *Proc. ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, Orlando FL, June 1994.

[35] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data caches and set-associative caches. In *Proc. Real-Time Technology and Applications Symposium*, pages 192–202, June 1997.