Precise WCET Calculation in highly variant Real-Time Systems

Pascal Montag
Daimler AG
pascal.montag@daimler.com

Sebastian Altmeyer
Saarland University
altmeyer@cs.uni-saarland.de

***Abstract*—Embedded hard real-time systems that are based on software product lines using dynamically derivable variants are prone to overestimations in static WCET analyses. This is due to the fact that infeasible paths in the code resulting from infeasible variant combinations are unknown to the analysis.**

This paper presents an approach to incorporate variant constraints in the calculation to exclude infeasible paths and thus to decrease the WCET overestimation. Based on feature models we propose a sound approach to identify significant infeasible paths that can be safely discarded in the analysis. The benefits of the approach are exemplified by a real world example from the automotive domain where we are able to reduce the WCET bound by up to 50 percent.

I. Introduction

Software is most valuable when it is reused in many products or product variants. In the embedded systems domain, software variants usually originate from different hardware combinations and configurations that are controlled by the embedded software. For example, car lines can be configured with several engine types, gear boxes and other different features. However, the same electronic control units (ECUs) are often used for all variants to decrease production costs and increase the efficiency in the garages.

The implementation of the resulting software variants ranges from the use of several EEPROM parameters to self-parameterization by detection of the hardware environment (similar to plug-and-play). Highly variant systems are characterized by an amount of variants that render the validation and verification of each single variant instance impossible. Hence, sophisticated methods for testing and analyses have to be applied to the entire ECU (including all possible variants).

In the case of embedded hard real-time systems a safe estimation of the worst-case execution time (WCET) is required for the verification of the entire system behavior. When applying static WCET analyses to highly variant systems, conservative assumptions can lead to significant overestimations [9].

Variants can be seen as system level constraints. For example, engines using a turbo charger require different handling of airflow and pressure control than engines without turbo chargers. However, due to system level constraints the combination of different airflow and pressure control functionalities is not possible. This mutual exclusion (cf. Figure 1 b) is often not taken into account by the WCET analysis. This is mainly due to the fact that system level constraints are set by external variables that are not visible to the analysis. A further reason for WCET overestimation arises from the necessity of context approximations by the WCET analysis. As it is practically impossible to trace each and every reachable program context, system wide dependencies are prone to be neglected by analysis optimizations. In this paper, we present

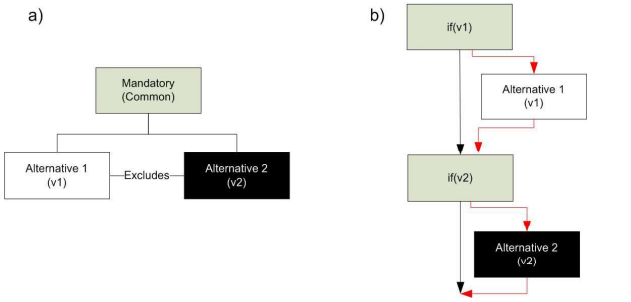

Fig. 1. a) Variant constraint model b) Variant algorithm with infeasible WCET path.

a variant-aware timing analysis, which derives a) a reduced but sound WCET bound and b) the corresponding worst-case variant that leads to this bound. The analysis consists of three steps. We first determine the variant dependent control-flow structures (loops and conditionals) and the type of dependency. Then we derive a set of candidates for the worst-case variant. In the last step, we extend the computation of the worst-case execution path from the original timing analysis to derive the WCET bound and the worst-case variant at once. Evaluation shows a reduction of the WCET bound by up to 50%.

In the remainder of this paper, we present the original timing analysis on which we base our approach in Section II. The formal setting of the variant-aware timing analysis is given in Section III and Section IV presents the analysis in detail. Evaluation is then given in Section V and related work in Section VI. Section VII concludes the paper.

II. Timing Analysis

We build our variant-aware timing analysis on top of the *aiT-Framework* as depicted in Figure 2. It consists of a set of different tools that can be subdivided into three main parts: *CFG Reconstruction*, *Static Analyses* and *Path Analyses*.

The *CFG reconstruction* builds the control-flow graph (CFG), the internal representation, out of the binary executable

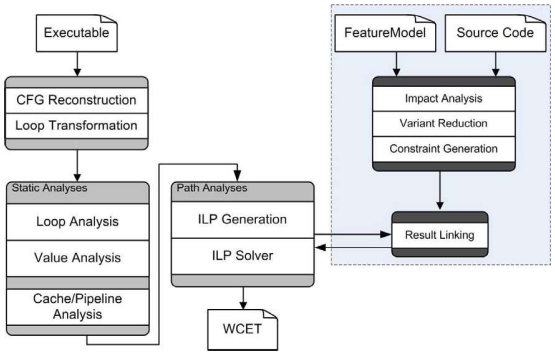

Fig. 2. The original aiT toolchain and the approach's extensions (in the dashed box)

[10]. This CFG consists of so-called *basic blocks*. A basic block is a sequence of instructions such that the basic block is always entered at the first and left at the last instruction. To make sophisticated interprocedural analysis techniques applicable, loop structures have to be transformed into tail-recursive routines. Additionally, user annotations, such as upper bounds on the number of loop iterations which the analysis cannot automatically derive, are processed during this step.

The static analysis part consists of three different analyses: *loop analysis*, *value analysis*, and a combined *cache and pipeline analysis*. The *value analysis* determines the effective addresses of memory accesses and also supports the loop analysis to find upper bounds on the number of loop iterations [8]. For this purpose, the analysis derives intervals for all variables at each program point.

The *loop analysis* collects invariants for all potential loop counters. This means it computes for all the variables changed within a loop, how much they change during one iteration. Then it evaluates the loop exits, requests start and end values for these potential loop counters from the value analysis and thus derives upper bounds on the number of loop iterations.

The *cache and pipeline analysis* performs the so-called *low-level analysis*. It simulates the processor's behavior in an abstract fashion to determine an upper bound on the execution time of each basic block[3], [5].

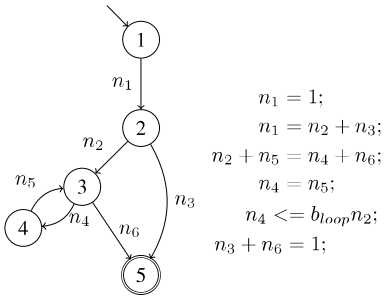

Fig. 3. Control flow graph and the corresponding flow constraints

The *path analysis* combines the timing information of each basic block and all loop bounds and searches for the longest path within the executable. In this fashion, it computes an upper bound on a task's execution time. Searching for the longest path is done by using a technique called *implicit path enumeration (IPET)* [6]: the control flow graph and the loop bounds are transformed into *flow constraints*. The upper bounds for the execution times of the basic blocks as computed in the cache and pipeline analysis are used as weights. Figure 3 provides an example. The variables n_i, also called traversal counts, denote how often a specific edge is traversed. The first and the last basic block are left, resp. entered, exactly once ($n_1 = 1$; and $n_3 + n_6 = 1$;). For all other basic blocks, the sum of the traversal counts entering equals the sum leaving. The loop body (basic block 4) is executed at most b_{loop} times as often as the loop is entered ($n_4 <= b_{loop} n_2$;). The constant c_j denotes the cost of the basic block j. The maximum sum over the costs of a basic block times the traversal counts entering it determines the final WCET bound.

III. Variant Constraint Specification

A common language for the description of variants is the feature model [1]. This modeling language describes abstract features of a product and the feature relationships as a tree. The abstract features are either mandatory (necessary in all products), optional (optionally available) or alternative (exactly one of a given set). These feature types form a basic constraint language that can be enhanced by more elaborate constraints.

A feature is anything that a customer can experience as a functional entity. Hence, an engine is a mandatory feature, whereas the engine type is an alternative decision. A sunroof is usually an optional feature, that can only be chosen if the car is not a convertible. Obviously, there are lots of similar constraints in the software that controls all these functionalities. While feature models are usually seen as

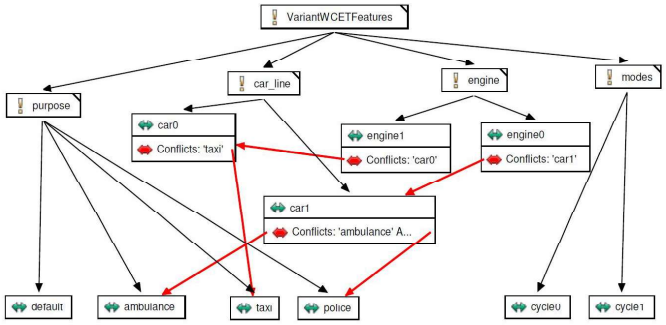

Fig. 4. An example feature model.

marketing decision models, they can be used further. For example, for the specification of the relationship of operating-modes. The decision-space of a feature model in current real-life applications may easily reach up to several million valid combinations. We now provide a formal definition of a variant.

Definition 1 (Variant)**.** *Given a set of features $\mathbb{F}$, a variant is a mapping from features to boolean values, depending on whether or not the feature is selected:*

$$V : \mathbb{F} \rightarrow \mathbb{B}$$

where $\mathbb{B}$ is the set of boolean values $true$ and $false$. If a variant obeys all constraints $\mathbb{C}$ determined by the variant

model, the variant is said to be valid. The set of all variants is given by $\mathbb{V}$ and the set of all valid variants by $\mathbb{V}_v \subseteq \mathbb{V}$.

Note that we do not need to consider mandatory, optional or alternative features separately as such requirements can be considered to be modeled by constraints contained in $\mathbb{C}$. For the sake of simplicity, we represent a variant V as a set of features that evaluate to true: $f \in V \Leftrightarrow V(f) = T$ and $f \notin V \Leftrightarrow V(f) = F$. Given the example in Figure 4, we have the following set of features $\mathbb{F}$ and set of constraints $\mathbb{C}$:

$$\mathbb{F} = \{car0, car1, engine0, engine1, cycle0, cycle1, default, ambulance, taxi, police\}$$

$$\begin{aligned}\mathbb{C} = \{&(car0 \otimes car1), (engine0 \otimes engine1), (cycle0 \otimes cycle1),\\ &\neg(car0 \wedge engine1), \neg(car1 \wedge engine0), \neg(car0 \wedge taxi),\\ &\neg(car1 \wedge police), \neg(car1 \wedge ambulance)\\ &(\exists! x \in \{default, taxi, police, ambulance\} : x)\}.\end{aligned}$$

For instance, $V = \{car1, engine1, taxi, cycle0\}$ is a valid variant. Note that within this small example there are already 2^{10} variants from which only 10 are valid.

Timing analysis may derive a variant-depending timing bound. Given a variant V, the specific information about all identifiers can be encoded to be considered by the timing analysis. The resulting time bound is then only valid for this variant V. In such a case, we write WCET(V). If no information about a specific variant or a set of variants is taken into account, we write $\widehat{\text{WCET}}$ for the variant independent time bound. Note that $\forall V \in \mathbb{V} : \widehat{\text{WCET}} \geq \text{WCET}(V)$ The problem of variant-aware timing analysis can then be seen as follows

$$\text{Find } \hat{V} \in \mathbb{V}_v \text{ s.t. } \forall V' \in \mathbb{V}_v : \text{WCET}(\hat{V}) \geq \text{WCET}(V')$$

This variant $\hat{V}$ is the so-called worst-case variant and provides a precise global bound for the analyzed software. Determining this variant by exhaustively deriving all WCET(V) is computationally infeasible due to the high number of different variants and the complexity of the timing analysis. Hence, we first reduce the search space and then aim for an approximate solution.

IV. Variant-Aware Timing Analysis

Our approach for a variant-aware static WCET analysis (cf. the dashed box in Figure 2) consists of the following steps:

- Impact Analysis: Which code parts are variant-dependent and deliver major impacts on the WCET?
- Variant Reduction: Which variants are relevant to the analysis? Which variants can be removed?
- Constraint Generation/Result Linking: The remaining variants are transformed into ILP control flow constraints. Hence, a variant-enriched ILP can be generated which is then used to estimate $WCET(\hat{V})$.

A. Impact Analysis

Two different control-flow structures may influence the timing behavior depending on the chosen variant: loops and conditionals. In case of conditionals, the taken branch may depend on whether or not a feature is selected. In case of loops, the loop iteration bound may be determined by the feature selection. To analyze these dependencies, we first need to identify the feature-dependent control-flow structures and in a second step analyze, how these structures are influenced. The set of identified conditionals is denoted as $\mathbb{CO}$ and the set of loops as $\mathbb{LO}$. Note that for the sake of simplicity, we restrict the description of the approach to *for*- and *while*-loops as well as to simple *if*-statements. Other statements can be seen as extensions to our basic structures. For example, switch-cases can be seen as combinations of multiple if-else structures.

We represent the variant-dependencies in the following way:

$$Cond : \mathbb{CO} \rightarrow (\mathbb{F} \rightharpoonup \mathbb{B})$$

A conditional c is assigned a partial mapping of features to boolean values. $(Cond(c))(f)$ evaluates to boolean value b, iff conditional c always evaluates to b in case feature f is selected.

$$Loop : \mathbb{LO} \rightarrow (\mathbb{F} \rightharpoonup \mathbb{N})$$

A loop l is assigned a partial mapping of features to integers. $(Loop(l))(f) = n$ means that loop l is bounded by n if feature f is selected.

More complex dependencies between features and conditionals are possible. Consider a conditional that evaluates to true, iff two out of three features are selected. Such cases do not fit into this notion. The domain of the partial mapping $(\mathbb{F} \rightharpoonup \mathbb{B})$ for such a conditional would be empty.

Listing 1. Variant Depending Code Fragment

```
    ...
11: const char clutch[2] =
12: /*  car0   car1 */
13:     {  10,    5 }; /*clutch line*/
    ... [Block1] ...
37:  for(i=0;
38:        i<clutch[c_car_type];
39:        ++i)
40: {
41:    if(c_purpose==TAXI){...[Block2]...}
    ...[Block3]...
94: }
    ...[Block4] ...
```

As a short example, let us consider the feature model in Figure 4 and the code snippet from Listing 1. In the example (stripped down from existing code) we notice that the dependency between feature model and code variables is not always given directly. Hence, these relationships have to be established via naming rules or manual mapping. In the above case, for example, prefix 'c' indicates a variant configuration parameter (*c_engine*, *c_car_type* and *c_purpose*). There are several possible ways to detect the basic structures in the code. We decided for pattern matching as the following step of constraint generation does not rely on a complete set of variant dependencies. Undetected dependencies on loops

and conditionals will lead to missing constraints and thus to an overapproximation of the result (but never to an underapproximation). The impact analysis first finds the loop in line $L37$ and the conditional in line $L41$ and then evaluates both expressions: $Loop(L37) = \{(car0, 10), (car1, 5)\}$ and $Cond(L41) = \{(taxi, true)\}$.

B. Variant Reduction

Given the functions $Cond$ and $Loop$, we can perform the variant reduction in order to reduce the search-space for our worst case variant.

1) Removing Timing-Irrelevant Features: We partition the set of features $\mathbb{F}$ into timing-relevant $\mathbb{F}_r$ and timing-irrelevant $\mathbb{F}_{\neg r}$. A feature f is timing-relevant if there is at least one loop l or conditional c such that $Cond(c)(f)$ or $Loop(l)(f)$ is defined. In our example, $\mathbb{F}_r = \{car0, car1, taxi\}$. Note that we also remove all constraints from the set $\mathbb{C}$ that contain at least one timing-irrelevant feature: $\mathbb{C}_r = \{(car0 \otimes car1), \neg(car0 \wedge taxi)\}$. Since the set of constraints only limits the possible variants, removing constraints may only lead to an overapproximation and is thus sound.

Definition 2 (Reduced Variant)**.** *A variant is called reduced, iff it only defines values for timing-relevant features $\mathbb{F}_r$:*

$$V_R : \mathbb{F}_r \to \mathbb{B}$$

The set of all reduced variants is denoted as $\mathbb{V}_r$.

The set of reduced variants for our example is

$$\mathbb{V}_r = \{\{\}, \{car0\}, \{car1\}, \{taxi\}, \{car0, car1\}, \{car0, taxi\}, \{car1, taxi\}, \{car0, car1, taxi\}\}.$$

The set $\mathbb{V}_{r,v} = \big\{\{car0\}, \{car1\}, \{car1, taxi\}\big\}$ gives all valid reduced variants. All other variants do not comply with $\mathbb{C}_r$.

2) Dominating Features: In order to reduce the set of possible worst-case variants even further, we need to identify dominant features with respect to timing.

Definition 3 (Feature Domination)**.** *A feature f is said to dominate feature f' ($f \sqsupseteq f'$), iff*

$$\forall c \in \mathbb{CO} : (Cond(c))(f) = (Cond(c))(f')$$

and

$$\forall l \in \mathbb{LO} : (Loop(l))(f) \geq (Loop(l))(f')$$

In our example, $engine0 \sqsupseteq engine1$ since the loop bound of the loop in line 38 is higher in case $engine0$ is selected (while all other loops and conditionals remain unchanged).

Using the feature domination, we can safely exclude some variants from the search space for the worst-case variant. Hence, we lift the domination relation to variants.

Definition 4 (Variant Domination)**.** *A variant V is said to dominate variant V' with $V \neq V'$, iff*

$$\forall f' \in V' : f' \in V \vee (\exists f \in V : f \sqsupseteq f')$$

In such a case, we write $V \sqsupseteq V'$

Assuming that a variant V is dominated by another variant V', we know that if V leads to the highest WCET-bound then also V' does. In our example variant $\{car0\}$ dominates $\{car1\}$, but not $\{car1, taxi\}$.

3) Variant Search Space: The variant search space $S \subseteq \mathbb{V}_{r,v}$ is a subset of the set of all reduced variants. In addition, we can exclude all dominated variants from the search space.

$$\forall V \in S : \nexists V' \in S : V' \neq V \wedge V' \sqsupseteq V$$

For the given example, the resulting search space is given by $S = \big\{\{car0\}\{car1, taxi\}\big\}$.

C. Constraint Generation / Result Linking

In the last step, we extend the IPET model to derive the worst-case variant.

Within the ILP to derive the worst-case path, we introduce one variable V_i for each remaining variant from the search space S and one variable f_j for each timing relevant feature from the set $\mathbb{F}_r$. Constraint (1) ensures that at most one variant is selected:

$$\sum_i v_i = 1 \tag{1}$$

The set of constraints (2) links the remaining variants (from the search space S) to their active features:

$$\forall_{f \in \mathbb{F}_r} : f = \sum_{V \in V^f} V \tag{2}$$

where $V^f = \{V \in S | f \in V\}$ is the set of variants where feature f is selected.

We now need to link the variants and features to the IPET model. For a feature-dependent loop $l \in \mathbb{LO}$, we add a constraint which bounds loop l by $(Loop(l))(f)$, in case feature f is selected:

$$\forall l \in \mathbb{LO} : \forall f \in Dom(Loop(l)) : n_{loop-body(l)} \leq (Loop(l))(f) \cdot n_{loop-entry(l)} + (1 - f) \cdot C_{big} \tag{3}$$

Constant $C_{big} \in \mathbb{N}$ is a big integer used to 'deactivate' constraint (3) if the corresponding feature is not selected. Given that C_{big} is chosen big enough, the constraint does not influence the final result. In the same manner, we add a constraint for each feature-dependent conditional $c \in \mathbb{CO}$. Let $(Cond(c))(f)$ evaluate to b. If feature f is active, traversal count for the $\neg b$ edge (denoted as $n_{\neg b}$) is set to 0.

$$\forall c \in \mathbb{CO} : \forall f \in Dom(Cond(l)) : n_{\neg(Cond(c))(f)} \leq (1 - f) \cdot C_{big} \tag{4}$$

Again $C_{big} \in \mathbb{N}$ is used to 'deactivate' the constraint, in case f is not selected.

Figure 5 depicts the control flow graph and the corresponding constraints for the code fragment from Listing 1.

The solution to the ILP determines the worst-case variant $\hat{v}$ and an upper bound on the execution time valid for this variant $WCET(\hat{v})$.

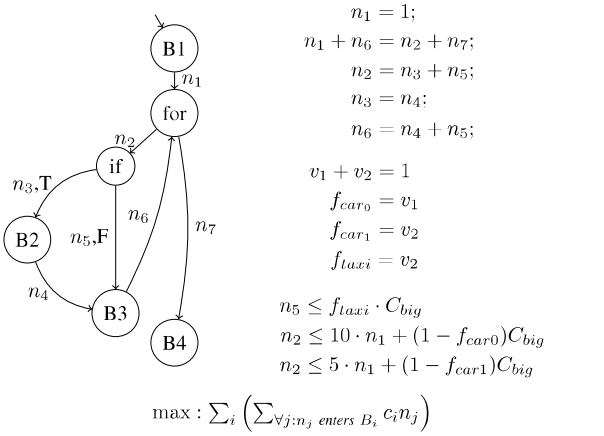

Fig. 5. Control flow graph and the corresponding flow constraints

D. Correctness

Each static WCET analysis can only estimate the actual worst-case execution time. It is, however, crucial that the actual execution time is never underestimated. The constraints introduced in Section IV-C model restrictions on the control flow graph and thus reduce the WCET bound. We now need to argue that this reduction is sound, i.e. that the WCET bound is still a safe approximation.

The allowed combinations of features (and thus of loop-bounds and conditionals) depend on the set of variants $\mathbb{V}_{r,v}$. Although the set of reduced and valid variants is much smaller than the set of variants $\mathbb{V}$, $\mathbb{V}_{r,v}$ represents an overapproximation of the $\mathbb{V}_v$ with respect to timing. By construction, concentrating only on timing-relevant features does not influence the WCET analysis. In step three, we also reduce the set of constraints $\mathbb{C}_r$. This, however, may only lead to variants falsely considered valid (less constraints lead to more valid combinations). Hence, we approximate the set of variants considered in the final ILP on the safe side only.

The constraints link loops $\mathbb{LO}$ and conditionals $\mathbb{CO}$ to features $\mathbb{F}_r$, which again are linked to reduced and valid variants $\mathbb{V}_{r,v}$. Both sets $\mathbb{LO}$ and $\mathbb{CO}$ only contain control flow structures for which it was possible to determine a feature dependency. Loops and conditionals for which we were unable to determine a dependency remain unchanged and are not influenced by our newly generated constraints. Again, we approximate only on the safe side and may only deliver an overapproximation, but no underapproximation.

Note that the derived worst-case variant $\hat{V}$ is only valid with respect to the timing analysis. It may happen that the actual worst-case execution time occurs for variant $V \neq \hat{V}$. However, the bounds computed by our analysis are still valid, i.e., $WCET(\hat{V}) \geq WCET(V)$ holds for all variants V, where $WCET(V)$ denotes the execution-time bound and not the actual worst-case execution time.

V. Evaluation

The introduced variant-aware WCET analysis was applied to several functions of the Daimler Trucks automatic gear shifting application (AG). The existing AG feature model was previously used for documentation- and test-case-generation purposes. An existing feature model should be the typical use case for the introduced approach so that no further effort is necessary for the description of the variant constraints. The used feature modeling tool is called *pure::variants* which is currently being rolled out in several Daimler projects.

The entire feature model consists of 72 features that lead to an estimated amount of 6.4 million valid variants, according to the modeling tool. A stepwise analysis of all single valid variants is thus not reasonable, which makes the AG a more than appropriate use case for our approach. For comparability reasons, we apply the variant-aware WCET analysis to three separate AG-functions (i.e. *ecomain*, *preparation* and *target_range*).

A. Impact Analysis

The input for the impact analysis is a simple mapping between features and variable names (in the C-code). In our case this manual work is rather simple as the variant-related code structures can be found in central parameterization header files where similar names are used for features and variables. This way, 60 out of 72 features are mapped to one or (less often) multiple variables. The remaining 12 features are either structural model elements or features without direct code impact. The automatic impact analysis applied to the entire software identifies 14 out of 122 loop counters that are variant-dependent (11 percent) while 499 of 3658 conditions are variant dependent (14 percent). Hence, if we assume a similar distribution for the other control flow statements (i.e. do-while, switch etc.), approximately every 10^{th} control flow decision depends on constant system constraints.

B. Variant Reduction

Our implementation of the variant reduction removes all features that are directly or indirectly irrelevant to the analyzed source code. Applying the reduction introduced in Section IV-B, only 7 features are directly relevant in the case of function *preparation* and 10 features in the cases of *ecomain* and *target_range*. Additionally, we implemented a dependency analysis that checks if two features exhibit a transitive dependency. Assuming that, for example, only the features *amount of forward gears* and *amount of backward gears* are used in the code, the additional feature *gear box* is required to identify the constraints of their transitive relationship.

The analysis reveals that 55 features are required to describe the dependencies among the 7 original features in case of function *preparation*. Respectively, 58 features are required for *ecomain* and *target_range*. We see that between 19% and 26% of the features of our feature model can be discarded from the analysis which might result in a decrease of analysis precision. However, the choice is between speed and precision while we always keep a sound result.

C. Constraint Generation and Solution

The remaining feature model, the dependent loops and conditions are then transformed into single ILPs per analyzed

function. The size of these ILPs ranges from 43 (*preparation*) to 61 (*target_range*) constraints. Before these ILPs can be solved, they are linked to the intermediate aiT result ILP.

Analyzing the function *preparation* without variant constraints reveals a WCET of 201 μs. Adding the variant constraints leads to a WCET of 193 μs, a reduction of almost 4 percent. In addition, the ILP solution also provides a worst-case variant which is the so-called *Unimog truck* using a particular engine and gear box. A portion of the 4 percent reduction can be traced back to the fact that many loops iterate over the amount of forward and backward gears. The gearboxes are available with up to 16 forward and 8 backward gears. However, a gearbox that exhibits both features at the same time is not supported by the software. Hence, a gearbox with 16 forward and 4 backward gears was automatically identified as the worst-case.

The function *target_range* was analyzed in a similar way. A variant-unaware WCET of 258 μs can be seen as highly overestimated when compared to the 129 μs variant-aware WCET. This is a total of 50 percent. Furthermore, the result analysis reveals a different 'worst-case truck' than before. The function's loops still iterate on the amount of gears, but the particular truck (with a maximum of 6 forward gears) requires special linearization functions to be called within each loop iteration which leads to the found worst-case.

Finally, also *ecomain* revealed an improvement of more than 3 percent. The main reason here can be seen in the extensive use of mode variables that trigger different functions in four separate execution cycles. The result allows us to identify the worst-case cycle and thus, to improve the distribution of functions within these cycles to gain a real WCET benefit.

Generally, we can see a very diverse range of potential reduction (3 to 50 precent) in the overestimation of WCET caused by software variability. An explanation for this result is the range of variant impact on the source code and thus the WCET itself. With the three use cases, our approach has proven to safely calculate reduced WCET estimates and to additionally deliver worst-case variants that were previously unknown.

VI. Related Work

To our best knowledge, the issue of variant-aware timing analysis has been hardly addressed this far.

The incorporation of *operating modes* in the WCET analysis (e.g. [4], [7], [11]) is strongly related to our approach: As in our approach, a given piece of software exhibits different execution times depending on a selected *mode* or *variant*, respectively. While variants are statically fixed and explicitly visible, modes are implicit in the source-code (e.g. start-up, main-function, failure) and may change dynamically. In addition, the number of modes is rather small, the number of valid variants may reach up to several thousands.

In [2], Ermedahl et al. presented a method to derive the input leading to the worst-case execution time. They perform a binary search over the value range of each input parameter. However, their approach does not handle boolean decisions variables and is thus not applicable to variant-aware timing analysis.

VII. Conclusion

This paper introduces a novel approach to incorporate system level variant constraints into the static worst-case execution time (WCET) analysis. To this end, the approach first identifies variant-dependent control flow structures, derives a set of potential worst-case variants and finally computes a reduced variant-dependent WCET bound and the corresponding worst-case variant. In addition, we have shown that despite more accuracy the method results in a sound overapproximation, i.e., the derived WCET bound can only over-, but never underapproximate the actual worst case execution time.

Based on industrial hard real-time examples, evaluation shows a timing bound reduction of up to 50%. However, as the amount of potential reduction mainly depends on the (unknown) amount of overestimation introduced by variability, our results cannot be evaluated by the achieved reduction rate. In contrast, our achievement is to enable this kind of reduction at all. Our examples show furthermore that the process scales well with complex and large variant spaces. This allows even more complex software and feature models to be analyzed with the presented methods. For the future, we plan to support a wider range of variant-dependent control flow structures to further increase the precision of our approach. In addition, we want to establish variant-dependent timing analysis in *Daimler*'s development process.

References

[1] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[2] A. Ermedahl, J. Fredriksson, J. Gustafsson, and P. Altenbernd. Deriving the worst-case execution time input values. In *21st Euromicro Conference of Real-Time Systems, (ECRTS'09)*, July 2009.

[3] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2-3), 1999.

[4] M.-L. Ji, J. Wang, S. Li, and Z.-C. Qi. Automated WCET analysis based on program modes. *The Computer Journal*, 52(5), 2009.

[5] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *Proceedings of the Static Analyses Symposium (SAS)*, volume 2477 of *LNCS*, Madrid, Spain, 2002.

[6] Y. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *in Proceedings of the 32nd ACM/IEEE Design Automation Conference*, 1995.

[7] P. Lucas, O. Parshin, and R. Wilhelm. Operating mode specific wcet analysis. In C. Seidner, editor, *Proceedings of the 3rd Junior Researcher Workshop on Real-Time Computing (JRWRTC)*, October 2009.

[8] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of loops. In K. Koskimies, editor, *CC*, volume 1383 of *Lecture Notes in Computer Science*. Springer, 1998.

[9] P. Montag, S. Görzig, and P. Levi. Challenges of timing verification tools in the automotive domain. In *2nd IEEE International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Paphos (Cyprus)*, November 2006.

[10] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7), July 2009.

[11] R. Wilhelm, P. Lucas, O. Parshin, L. Tan, and B. Wachter. Improving the precision of WCET analysis by input constraints and model-derived flow constraints. In S. Chakraborty and J. Eberspächer, editors, *Advances in Real-Time Systems*, LNCS. Springer-Verlag, 2010. To appear.