Applying Static Timing Analysis to Component Architectures

Pascal Montag and Steffen Görzig
DaimlerChrysler Research and Technology
Department REI/SC
Wilhelm-Runge-Strasse 11, 89081 Ulm,
Germany

firstname.lastname@daimlerchrysler.com

Paul Levi
Stuttgart University
Institute for Parallel and Distributed Systems
Universitätsstrasse 38, 70569 Stuttgart,
Germany

paul.levi@informatik.uni-stuttgart.de

ABSTRACT

The increase in software functions and software complexity of automotive applications requires appropriate software architectures. A promising approach is the component architecture which also stands in the centre of the automotive standardisation project AUTOSAR[2].

As every embedded real-time system inevitably has upper time bounds, we present an integrated method of timing estimation for highly flexible and variant applications based on a prototype component architecture. Therefore, we especially develop methods for parameterised timing estimation which depend on the grade of complexity, variability and necessary exactness.

The feasibility of the introduced concept is shown in the prototype architecture and a prototype application.

Categories and Subject Descriptors

B.8.2 [Performance and Reliability]: Performance Analysis and Design Aids; D.2.11 [Software Architectures]: Domain-specific architectures

General Terms

Performance, Verification

Keywords

WCET Analysis, Parameterised Timing Analysis, Automotive Component Architectures

1. INTRODUCTION

Today's cars contain embedded control units (ECU) for controlling the engine, the power train, the door locks, the wind screen wiper and several other components. While these functions are included in nearly every car, other features (e.g. driver assistance features) will be increasingly optional or parameterisable. A system-wide software architecture

which handles the arising complexity and variability is required. Taking into account that automotive software is mostly developed by suppliers (mainly basic software for all cars) and the OEM¹ (mainly special features) together, this software architecture should consequently be used by all major car manufacturers. The *Automotive Open Software Architecture* (AUTOSAR [2]) which is basically a component architecture is an approach aiming at this goal.

The development of applications based on such an architecture is done by adapting and combining newly developed and existing components. The applications run on all the platforms which are compatible to the architecture and meet the requirements of the application.

The application requirements are mainly derived from non-functional aspects (e.g. communication hardware, processor speed or time overhead of the platform). The most difficult non-functional requirement to determine is the time consumption.

The *worst-case execution time* (WCET) forms the basis of all timing verification. This value has to be estimated for the software-platform (operating system overhead), the architecture (component management) and all components.

As timing properties vary according to the parameters of the components (e.g. the number of gears for an automatic gear shifting component) and the platform on which they are executed, analysis results are rarely single values for the components.

The aim of this paper is to determine the variation and dependencies which influence the timing calculation. Suitable methods will be developed which can be applied depending on the complexity of the variation.

2. TIMING ANALYSIS OVERVIEW

As the estimation of the whole system timing behaviour is not a trivial task, the analysis activities are usually employed at three different levels.

1. Single Task Level: Single, not pre-empted tasks or routines are analysed on specific hardware platforms. The result of this analysis is the worst-case execution time (WCET).
2. ECU Level: The embedded control unit (ECU) represents a complete system which consists of a platform software (low level drivers, operating system, etc.) and

¹Original Equipment Manufacturer

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SEAS'06, May 23, 2006, Shanghai, China.

Copyright 2006 ACM 1-59593-085-X/06/0005 ...\$5.00.

functional software components. Important aspects of the analysis on this level are task-scheduling (strategy, priorities, etc.) and interrupt behaviour (interrupt handlers, priorities, etc.). The result of this analysis is the worst-case response time (WCRT) for all tasks and interrupt service routines (ISRs).

3. System Level: Today the typical automotive software system is distributed. On this level, timing properties and dependencies of single ECUs are analysed with regard to their communication infrastructure (kind of distribution, communication hardware, protocols, etc.).

The focus of this paper is the calculation of the WCET on the single task level. The ECU level calculation which is performed by a schedulability analysis is based on this WCET estimation. Similar to this, the system level is analysed by using the results of the ECU level.

2.1 Static WCET Analysis

Time consumption occurs within information processing systems when the hardware executes single commands and command sequences. That is why a static analysis is always based on two views of a system: the software and the hardware view.

The software view concentrates on finding possible execution paths and excluding impossible execution paths. Doing this exactly is impossible since the halting problem is undecidable. Therefore, some overestimation of the execution time is unavoidable. To estimate an execution time which is not equal to infinity, at least upper bounds for the iteration numbers of all loops have to be known. These values can be retrieved manually or automatically by analysis methods like the abstract interpretation [4]. To calculate a more precise time estimation, all control flow conditions have to be known. Therefore, it is necessary to know not only all possible values of flow relevant variables but also the execution context in which specific values may occur.

The second necessary view is the hardware view. Here the software is linked to a special hardware platform on which it is executed. Due to today's hardware complexity, the time used for a command sequence does not necessarily have to correlate with the sum of the atomic instruction execution times. This effect results mainly from CPU-features like caches and pipelines. In order to be able to use such hardware features, their timing benefits have to be taken into account while attaching atomic times to instructions depending on the context of their execution.

When all possible execution paths are connected with atomic time consumptions, the calculation of a maximum path can be done for example by solving an ILP (Integer Linear Program) which is the result of the Implicit Path Enumeration (IPE) [6].

The major advantage of static timing analysis compared to measuring methods is its safety (Figure 1). It is achieved by considering all possible paths in the calculation. That leads to an estimation which is higher than or at most equal to the real WCET. In contrast, measuring methods result in values which are lower than or at most equal to the real WCET. That is why safety can only be guaranteed by static timing analysis.

Until now measurement methods have been used more often than static timing analysis in most industries apart from safety relevant domains like aeronautics and space. That is

Figure 1: The difference between measuring and static analysis methods.

Figure 2: Static WCET estimation with aiT.

why there are only few studies on industry software at the moment [3]. This situation might change with the development of time triggered systems which will reach the market through future by-wire-systems [7].

2.2 AbsInt aiT

aiT is a static worst-case execution time analysis tool which uses the above-mentioned methods. It is developed by *AbsInt Angewandte Informatik GmbH* [1] which is a spin-off company of the *Universitt des Saarlandes*.

As timing analyses are always platform dependent, analyses with aiT can only be employed if the used processor is supported. aiT is available for a wide range of processors used in industry, like C166/ST10, TMS320C33, ARM7, HCS12/Star12, PowerPC 555/565/755 and ColdFire 5307.

The analysing program takes a binary input file which is disassembled and analysed as shown in figure 2. In this figure one can see that the results of the value analysis are not used iteratively in the loop bound analysis. Therefore, no values of loop constraints are known in the loop bound analysis and only simple loops can be found by pattern analyses. This leads to a low loop bound detection rate of the tool and the need for user annotations.

The reduction of the WCET overestimation also requires additional user annotated constraints in the AbsInt specification language AIS [5] which can be provided in an external file or directly in the source code. A typical annotation specifies a program point or memory address and the corresponding properties. The example

```
LOOP "_r1" + 2 loop begin max 3;
```

specifies that the second loop in routine "r1" (program point) has its loop condition at the beginning and is executed three times at the maximum (properties). The most important annotations are

- *Loop Bounds* specify an upper bound for the maximum number of loop cycles. They are necessary to receive

results at all. aiT automatically finds some loops. The remaining ones must be specified manually.

- *Flow Constraints* specify the relations between the execution counts of program points. Loop bounds, for example, are relations between program points within the loop and program points in the loop encapsulating block. Therefore, if a loop cycles three times, the relation between the loop body and the loop constraint is three to one.
- *Call Targets* are necessary if function pointers are used because aiT is not able to resolve them automatically.
- *Memory Accesses*. aiT tries to find the addresses of memory accesses automatically, but sometimes only finds an interval containing the real addresses, or finds no information at all. Specifying correct memory accesses leads to two positive results: If an access is unknown, the slowest memory is chosen out of the different kinds of available memory to calculate the time consumption of the access. Additionally, this prevents wide memory write accesses, which would destroy the value information of the variables situated in these ranges.

Timing analyses based on aiT have been applied successfully to a range of power train functions of Mercedes Benz trucks. Successfully means that the WCET estimations were close to the maximum measured values even though single functional entities had code sizes of 20 kLOC. Bigger deviations occurred only in functions which used floating point operations.

2.3 Component Architectures

Component Architectures as we use them have a layered structure as shown in figure 3. Each software component (functional entity) has a fixed set of routines for activating, running and deactivating the component.

A specific component is defined by its data interface which in turn is defined formally and stored in a common component repository.

The component repository, which stores all necessary information of the components, forms the basis of the application creation process. The construction of applications is done by selecting the demanded components and modelling the execution order and the communication between them. The modelling process also includes the definition of tasks and the distribution over several host processing systems. This is achieved by assigning the components to special component containers like tasks (on ECUs) or threads (on x86 platforms). These containers can be assigned to processes and these to hosts.

The application modelling process results in a formal description of the application (e.g. XML). Using this application description, the modelled system is transformed into a configuration description for the platform and the Run-Time Environment (RTE). The RTE handles the communication between the components and triggers off their execution.

The runtime environment is always specific to a software platform because it abstracts from a special operating system, while the operating system itself abstracts from a given hardware platform.

Figure 3: Simplified structure of a component architecture.

3. THE PROTOTYPE ARCHITECTURE AND APPLICATION

A prototype of a standard component system is implemented with regard to the special needs of embedded real time systems. In addition, a tool for a graphical application design is implemented which makes it easy to assign, for example, components to *RTE-cores* which represent an abstraction of tasks (e.g. on ECUs) or threads (e.g. on x86 architectures).

The function layout inside the RTE-cores is defined by component chains (sequentially executed components) which can be structured as graphs (by defining predecessors and successors). All RTE-cores of a host have a common data repository so that the components can communicate across task borders. Due to the fact that this could possibly lead to deadlocks, a locking mechanism for data entities is used.

An additional feature of our component architecture is that an application can assume different states which are defined by component activity vectors, so that a state is simply defined by a set of active and inactive components. The necessary state switches are caused by special decision components.

Several generators transform the application description into a specific operating system description of the tasks and interrupts used (e.g. OSEK OIL file). Furthermore, RTE instances are generated which are executed in the previously defined tasks. Thus, a running application can easily be distributed on several different platforms in several different configurations.

As a prototype application for this component architecture, a TSP² solver is implemented. It consists of three components: TSP_Client (optimises a given route), TSP_Server (sends results on the communication bus) and TSP_Admin (if a defined optimisation goal is reached it switches the TSP_Client off). The application is parameterised by the number of cities which are taken out of a given map.

²TSP = Travelling Salesman Problem

4. APPLYING TIMING ANALYSIS

The advantage of the component system is the flexibility in reusing components in different applications and the flexibility in distributing the application parts on several tasks and hosts.

The reason to change the structure and distribution of a running application is very often the failure in meeting the resource restrictions. This occurs especially when extensions have been applied to the components. As time measurements of each new application are unsafe and time consuming, the component application represented in the modelling tool has to be used to verify the restrictions.

4.1 Main Concept

The speciality of the component architecture concerning the timing verification is the high grade of variability which originates from distributing the components and the ability to parameterise the components. On the other hand, the static timing analysis takes advantage of the development infrastructure because the execution contexts of all components are more or less known.

The following subsections describe the analysis of the platform, the runtime environment and the components. In subsection 4.5, we show how these single analysis parts are combined to form a complete task WCET analysis.

4.2 Platform Analysis

In the first place, platform services which are executed independently by the operating system make contributions to the WCET. This leads to task offsets (e.g. release times) and time contributions by interrupt service routines (ISRs). Due to high complexity and hardware dependency, task offsets have to be evaluated by the platform supplier. In our analysis they are assumed to be zero. As we do not intend to analyse concurrency mechanisms, WCET estimations of the used ISRs are only calculated but not integrated at this level. This is done later by a worst-case response time analysis. The WCET-analysis for all used ISRs is performed easily since these routines are quite simple and do not need AIS-annotations.

A second platform time contribution occurs due to operating system calls from the RTE. They will be considered in the following RTE Analysis.

4.3 RTE Analysis

An analysis of the core function of the RTE with aiT needs six loop annotations and several standard annotations like the processor clock rate. The given maximum loop bounds depend on the maximum number of components in a component chain, the total number of component chains and the maximum number of data interface entities of a component. As aiT is not able to calculate the loop bounds automatically, it is also not able to calculate the context dependent loop bounds. Furthermore, it is not possible to annotate context dependent constraints. Therefore, the difference between the WCET estimate and the real WCET increases with the distance between the average number of components per RTE-core or data entities per component and their maximum values. The reason for this behaviour is that the same loop takes the worst case loop bound in all contexts. To avoid this problem, an analytical analysis method using aiT is employed for the RTE-core. Therefore, each single routine which is called by the RTE_core_cycle has to be

analysed separately. The aiT visualization of this routine is displayed in figure 4.

A single routine is analysed by removing the time consumption of the functions called by the routine and by changing the single loop bounds of the routine's loops step by step. The annotation for excluding routines from the analysis is "*snippet "_R" is not analyzed ...*". By using this method a linear dependency upon the loop bounds can be derived, which has an offset and a linear cycle factor. The timing calculation result for our prototype looks as follows:

$$t_{Core} = t_{CoreO} + k \cdot (t_{CoreC} + t_{ChainO}) + \sum_{i \in l} t_{ChainC} + t_{Ci} \quad (1)$$

$$t_{Ci} = 2 \cdot (t_{DataO} + m_i \cdot t_{DataC}) \quad (2)$$

The time consumed by the RTE-core routines (t_{Core}) results from the core routine offset (t_{CoreO}) and the number of executed component chains k . The component chains are executed in a loop which has its own cycle time (t_{CoreC}). The execution of the chains is realised by a loop which has an additional offset (t_{ChainO}). Furthermore, each component takes t_{ChainC} time per cycle.

Each data entity of a component is locked before and unlocked after an activity. That leads to an offset (t_{DataO}) and a cycle time for each data entity (t_{DataC}).

All times t_{CoreO} , t_{CoreC} , t_{ChainO} , t_{ChainC} , t_{DataO} and t_{DataC} can be calculated with aiT as constant values in CPU-cycles. The only variable values are k , the number of component chains, and m_i , the number of data entities for each of the l components. These variables can be easily derived from the application description.

When performing this kind of analytical analysis with aiT, it is important that no values which are changed in the annotations are fixed in the binary or otherwise available to aiT. Complex flow dependencies (e.g. by nested loops) and variable dependencies (dependencies which are estimated by aiT) have to be considered separately.

The RTE analysis is platform dependent because operating system routines are called from the RTE and the analysis is done for a special hardware platform with a special hardware configuration (e.g. activation of buses).

4.4 Component Analysis

The interface routines for all components have to be analysed separately. The TSP_Server and TSP_Admin routines do not need annotations to be analysed. But the TSP_Client requires eight loop annotations to achieve results. Two loop bounds are found by aiT automatically.

The annotations are given in an AIS-file. As loops are defined relatively by counting starting from the routine entry (e.g. routine_x + 3 loops), it is important to know that this is done on machine code level. As a result of this, in nested while loops the inner loops are possibly counted prior to the outer loop depending on the compiler. The correct counting can be ensured by analysing the visualisation or by using source code annotations.

To improve the result, two flow annotations were made to describe the dependency between two nested inner loops and the outer loop. The sum of the inner loop execution count and the outer loop execution count has to be the number of cities which is annotated by using the following AIS line:

```
FLOW ($IAddress1)+($IAddress2)<=$Number;
```

Figure 4: Call graph of the RTE-Core functions.

Method	Degree of Variability	Calculation Speed	Calculation Time for Prototype Component	Error-Prone
Fixed Values	no variability	zero	0 s	low
Manual Analytic	small	high	10^{-7} s	high
ILP Analytic	medium	medium	17 s	medium
aiT Analysis	high	low	54 s	low

Table 1: Time determination method comparison.

This annotation states that the sum of the execution count of the instructions at the memory address \$IAddress1 (e.g. an instruction in an inner loop) and at the memory address \$IAddress2 (e.g. an instruction in the outer loop) is \$Number (e.g. the number of cities). These flow constraints allow the analyser to remove the loop bound annotations of the inner loops even though this is not necessary.

A problem arises when this kind of annotation is applied to loops. In this case absolute memory addresses have to be used. If components are linked to a new binary application, these addresses will probably change which requires the annotations to be adapted. Two solutions to this problem are possible: either the annotations are changed to relative statements the way they are used for loop bound definitions or the analyses are made analytically.

While the first solution can be integrated with small effort (and will be done by AbsInt), the second solution has the additional advantage of faster and tool independent calculations. But analytical analyses, the way they are applied to the RTE-core, are not always suitable. Especially for highly complex components with a lot of non-linear dependencies such a manual analysis is error-prone.

That is why we introduce a second method which is able to handle more complex dependencies. As Integer Linear Programs (ILP) are used to describe the program flow dependencies, we make use of parameterised ILPs. An initial ILP can be found by using aiT, where it is generated as an intermediate result.

All manual constraints, given by the AIS-annotations, should include the complete variability of a specific component. This is the only way to avoid that aiT includes non-parameterisable variant information found in the binary code of a specific variant. To guarantee complete coverage of the variable component information in the manual constraints, this information can be transformed from the repository into AIS-annotations. Afterwards, this information can be found in one or multiple ILP-constraints which are of the form $ax_1 + bx_2 \leq cx_3$. These constraints can be detected by analysing the difference between the ILPs of two variants. This allows the analyser to parameterise the ILP and to solve it for all kinds of parameterised components.

We were able to show that this method works well for our TSP-prototype with quadratic loop dependencies. The parameter of change for all manual annotations depends on the number of cities which are on the route. The results for 30 cities in table 2 were derived for a 40MHz C167 processor by using a parameterised ILP on the one hand and by using aiT directly on the other hand. The activation and deactivation routines were not used in the prototype application. So all of them have the same WCET.

The change of single ILP-constraints is only valid in special cases which depend on the pipeline analysis. The result turns out to be correct if the initial ILP is derived from a component where all loops terminate at least after a defined number of iterations. If we use this ILP for the timing calculation of the parameterised component where the loops

Routine Component	Activation [ms]	Cycle [ms]	Deactivation [ms]
TSP_Client	0.00028	10.952	0.00028
TSP_Server	0.00028	0.0471	0.00028
TSP_Admin	0.00028	0.00355	0.00028

Table 2: Upper time bounds for the TSP-Components (30 cities).

turn less often than the defined number, the result is slightly overestimated. But as long as the loop body does not contain pointer operations which cause loop bound dependent memory area exceedings, the result should still be valid as it is never underestimated.

In need of tight WCET estimations all parameterised components have to be verified by aiT. This can be done by generating parameterised AIS-annotations from the component parameter description and using the batch mode of aiT. The disadvantage of this method is that it takes (much) longer than the other methods (see section 4.5) and requires the aiT tool. A comparison between the three methods can be seen in table 1. Finally we have four classes of time estimation for components which are stored in the component repository: Fixed Values (there are no dependencies or the dependencies are set to fixed values), Formula (the dependencies are linear and simple), ILP (the dependencies are complex and overestimations are not problematic) and AIS (the dependencies are complex).

The component repository is able to contain fixed values for special configurations of a component and a parameterised Formula, ILP or AIS file for all configurations which are not analysed. The application generator checks whether a configuration is available for the selected platform. If that is not the case it chooses the proper method to estimate the correct value and finally writes this value back to the component repository.

4.5 Combining the results

The resulting calculations for all application tasks based on the architecture are derived from formula (1). This formula was determined manually because the RTE functions are indispensable to the overall calculation and depend only linearly on three variables, which means that they are quite simple to determine.

Using this formula, our application has an RTE-offset of 0.243 ms (0.369 ms by using aiT directly which causes higher overestimations). The WCET of the whole task equals the sum of this value and the maximum execution times of all used components according to $\max(Activation + Cycle, Deactivation)$. This term results from the fact, that a component is executed in the same cycle in which it has been activated while a component is not executed after deactivation.

The task result of 449847 cycles or 11.24 ms was calculated in a few micro seconds by using the analytical method. In comparison, it took 17 s to estimate this WCET using the TSPClient ILP and 54 s using aiT for the TSPClient component.

Calculating the WCET of the whole application (RTE and components) with aiT is possible but takes more than 24 hours. This complete application analysis needs three additional flow constraints and six additional call target speci-

fications. The latter result from the component calls which are implemented as function pointer calls. The enormous difference in calculation speed, even for a small application, shows the need of scaleable calculation methods as they are presented here.

5. CONCLUSIONS AND FUTURE WORK

In this paper we presented a method for combining the static timing analysis with component architectures for real time systems. We were able to show that the static timing analysis takes advantage of the component architecture's infrastructure.

The static timing analysis was carried out by using aiT by AbsInt [1]. The main disadvantage of this tool is the low loop bound detection accuracy. Therefore, we are actually working on a source code based loop bound detection which generates the AIS (AbsInt Specification) annotations. By using a practical example we demonstrated how to integrate several calculation methods on the different levels of the architecture. Three approaches with different advantages and disadvantages to determine the worst-case execution time (WCET) of functions with timing variance were introduced. The analytical method turned out to be even more precise than the aiT tool itself because we are able to take into account the contexts which we know from the architecture. The additionally presented methods which use an Integer Linear Program (ILP) or the AbsInt Specification (AIS) introduced a certain amount of scalability into the calculation. Depending on the grade of variability of the analysed component and the grade of necessary exactness of the calculation a proper method may be chosen.

The combination of calculation methods increased the calculation speed for a small prototype application from more than 24 hours to 17 seconds (by combining the analytical and the ILP method). This advantage in conjunction with a given development process based on component architectures will probably increase the acceptance of the time verification technology in the industry.

Timing analyses of powertrain software components have shown the general possibility of applying the introduced methods to more complex components as expected in real world scenarios. However, these analyses are not a subject of this paper.

Making use of the results of the WCET estimations for all relevant system parts we are now able to analyse concurrency and distribution related time dependencies. These future goals will lead to an integrated software development and verification platform where most estimation tasks are done automatically.

The final target is an optimisation based on an application model or a set of application models and other external constraints (e.g. economical constraints) to determine an optimum hardware platform for the selected application.

6. REFERENCES

- [1] *AbsInt company homepage*. www.absint.com.
- [2] *AUTOSAR consortium homepage*. www.autosar.org.
- [3] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. Applying static WCET analysis to automotive communication software. In *17th Euromicro Conference of Real-Time Systems, (ECRTS'05)*, Mallorca, Spain, July 2005.

- [4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, Los Angeles, January 1977.
- [5] C. Ferdianand, R. Heckmann, and H. Theiling. Convenient user annotations for a WCET tool. In *3rd International Workshop on Worst-Case Execution Time Analysis*, 2003.
- [6] Y. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.
- [7] T. Ringler. Entwicklung und Analyse zeitgesteuerter Systeme. *Automatisierungstechnik*, (52):189..197, April 2004.