Towards Model-Driven Development of Hard Real-Time Systems

Integrating ASCET and aiT/StackAnalyzer

Christian Ferdinand[1], Reinhold Heckmann[1], Hans-Jörg Wolff[2], Christian Renz[2], Oleg Parshin[3], and Reinhard Wilhelm[3]

[1] AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
info@absint.com, http://www.absint.com
[2] ETAS GmbH, Borsigstraße 14, D-70469 Stuttgart, Germany
Christian.Renz@etas.de, http://www.etasgroup.com
[3] Universität des Saarlandes, Postfach 15 11 50, D-66041 Saarbrücken, Germany
wilhelm@cs.uni-sb.de, http://rw4.cs.uni-sb.de

Abstract Software developers in the automotive sector must achieve high quality objectives. Many design and implementation errors are avoided by synthesizing code from model-based software specifications using automatic code generators such as ETAS' **ASCET**. To verify non-functional properties of the implementation, model-based design processes should be complemented with static program analysis tools like **AbsInt**'s **StackAnalyzer** and timing analyzer **aiT**. **ASCET**, **StackAnalyzer** and **aiT** can be integrated in a way that the **aiT/StackAnalyzer** analysis results for code generated by **ASCET** are conveniently accessible from within the **ASCET** development environment. This gives **ASCET** users a direct feedback on the effects of their design decisions on resource usage, allowing them to select more efficient designs and implementation methods. In the paper, we present the tools, the experimental integration, preliminary results and plans for further tool integration.

1 Introduction

Software developers in the automotive sector face some specific challenges: Many software systems are safety-critical and, thus, must achieve high quality objectives. On the other hand, competitive markets require software and hardware that can be mass-produced using a minimum of resources. Additionally, today's cars feature complete networks of Electronic Control Units (ECUs), which require highly collaborative software development. Therefore, even from the start, safety and budget considerations influence the design and specification of automotive software systems.

Often these challenges induce conflicting goals concerning reduction of component costs, of development costs, and of development complexity. One example would be reduction of component costs by using cheaper ECUs without floating

point units, where all calculations need to be performed using integers. However, due to the additional scaling and converting needed due to the representation of floating point numbers by integers, this can become an additional source of defects during the actual implementation of the system, leading to an increase in development costs and complexity.

There are different approaches to deal with the development problems in the automotive context. Standards like MISRA-C [1] attempt to minimize the amount of errors introduced by manual coding. Following such guidelines, however, may increase the time needed for coding and incur a cost in higher resource usage.

Model-based design tries to satisfy the high safety requirements in combination with good development productivity by starting with a software specification. The implementation process is not necessarily automatic. It is therefore still possible to introduce software defects through misinterpretation of design and specification documents or through human error during the manual coding process. Automatic code generators such as the one provided by **ASCET** are increasingly used to generate the implementation from the specification. By creating C code directly from the model-based specification, these code generators avoid the typical translation problems that occur in the implementation stage.

Many design and implementation errors are avoided by synthesizing code from specifications. However, non-functional properties such as absence of memory overflow and timer overruns are still an issue. To verify such properties of the implementation, unit tests and runtime measurements are currently used in the industry. Assuming sufficient test coverage of the system, some information about the typical runtimes of the software processes can be obtained. However, to acquire a higher level of confidence and to aid the development process, it is necessary to gather reliable and precise information about the code. Recent advances in the area of static program analysis based on abstract interpretation led to the development of tools to automatically detect upper bounds on resource usage like worst-case execution times (WCET) and worst-case stack usage, and of tools to prove the absence of runtime errors like null pointer dereferencing and out-of-bounds array accesses.

When using automatic code generation, tools checking for runtime errors are of minor importance – the code generator is expected to produce correct code given its knowledge about the model. Tools to determine safe and precise bounds on resource usage, however, can be very helpful for the users of modeling and automatic code generation tools. Tools of this kind include **AbsInt**'s **StackAnalyzer** and timing analyzer **aiT**. Other timing tools, including academic prototypes, are described and discussed in [2].

In the context of safety-critical hard real-time applications, the standard use of tools like **aiT** and **StackAnalyzer** is to demonstrate and prove that pieces of code are guaranteed to always execute within limited time intervals and resource bounds.

In our work, we propose to complement model-based design processes with static program analysis tools. This guarantees the satisfaction of safety require-

ments, and it helps to speed up the project by aiding in the establishment of general guidelines, the configuration of the build environment used as well as the coordination of distributed development and the development itself. We argue that to develop hard real-time systems, model-driven development coupled with detailed analysis of the implemented software is much better suited than traditional development methods that rely on programming C code.

The users of **ASCET** usually work on a much more abstract level than the producers of manual code. **ASCET**, **StackAnalyzer** and **aiT** can be integrated in a way that the **aiT**/**StackAnalyzer** analysis results for code generated by **ASCET** are conveniently accessible from within the **ASCET** development environment. This gives **ASCET** users a direct feedback on the effects of their design decisions on the resource usage, allowing them to select more efficient designs and implementation methods.

In the following, we present the tools, the experimental integration, preliminary results and plans for further tool integration.

2 Model-Based Design and Automatic Code Generation

In the automotive industry, model-based design has rapidly become a standard technology for system development. Complex automotive functions are usually based on abstract function models that make use of domain-specific knowledge. In this context, model-based CASE tools can offer significant development benefits as they allow for an easier transfer of domain-specific knowledge into a software engineering context. A comprehensive study of different model-based CASE tools can be found in [3].

2.1 Model-Based Development for Real-Time Applications

Software development using C offers many degrees of freedom that make it more difficult to verify the fulfillment of safety and real-time requirements. It is therefore necessary to provide a more abstract and more clearly defined specification of the system to be developed.

To solve this problem, ETAS' **ASCET** offers graphical specification editors to model control and data flow and state machines for state-based algorithms, as well as textual specification using **ESDL**, a programming language with a syntax based on Java that operates on the model level. Working with these specifications allows the developers to abstract away from the concrete variables on the target and deal with (physical) model variables instead, each with a well-defined representation in terms of concrete variables. These specifications can then be used to generate C code both for rapid-prototyping as well as ECU targets. The code generator will take care of the translation of model variables to program variables (according to the chosen representation of model variables) and of implementing operations on the model variables in a way that is consistent with their concrete representation, thereby eliminating a lot of possible

oversights on the side of the developer. The code generators intended for producing C code used in series production were the first world-wide to be certified according to IEC 61508, the international standard for "functional safety of electrical/electronic/programmable electronic safety-related systems".

The improvements offered by this approach are demonstrated in [4]. To verify the correctness of their active steering model, BMW developed a formal verification tool that operates on components developed using **ASCET**.

To reduce the effort needed for verification, **ASCET** strongly supports modular development through so-called classes, encapsulated modules closely related to object-oriented programming concepts (while avoiding dynamic memory allocation and inheritance). Components can be reused by using multiple data sets and implementations depending on the project context. **ASCET** offers strong separation of algorithm, data and implementation details (memory classes, types, etc.), thus facilitating the software engineering process and the verification and testing process.

These improvements are especially useful in the context of fixed-point integer calculations on low-cost platforms, where manual coding typically introduces many bugs that can be avoided using code generation. To verify the model during different stages of development, **ASCET** offers several code generators to aid the developer in a step-by-step transition from model to production code. This allows verification of the code against the model on the PC using PC simulation of the generated code, and on real time-capable rapid-prototyping hardware similar to the target platform, but with additional resources. Finally the code can also be run on the target platform, either as a complete model or (using bypassing) as newly developed functions integrated into already released versions of the software.

2.2 Model-Based Development in the Context of Large Applications

To be usable for large-scale automotive applications, model-based tools need to integrate themselves tightly into existing toolchains. Amongst different tools, ANSI C code has established itself as a quasi standard for embedded development. Since **ASCET** allows for the integration both of models developed using Matlab/Simulink as well as legacy C code, we focused our analysis on compiled binaries as well as annotated C code.

3 Code Performance in Real-Time Systems

3.1 Stack Usage

Stack overflow is a possible cause of catastrophic failure that usually leads to run-time errors that are difficult to diagnose. The problems stem from the fact that the user needs to specify the amount of memory that should be reserved for the stack. Underestimating the maximum stack usage leads to stack overflow and thus system failure, overestimating means wasting valuable memory resources.

One approach to solve this problem is to measure the maximum stack usage using a debugger. However, even when running the program several times using a test suite, it is not guaranteed that the maximum stack usage is ever observed.

AbsInt's **StackAnalyzer** is able to provide a general worst-case estimate. By performing a value analysis on the stack pointer, the tool can figure out how the stack increases and decreases along all possible control-flow paths. This information can be used to derive the maximum stack usage of a task.

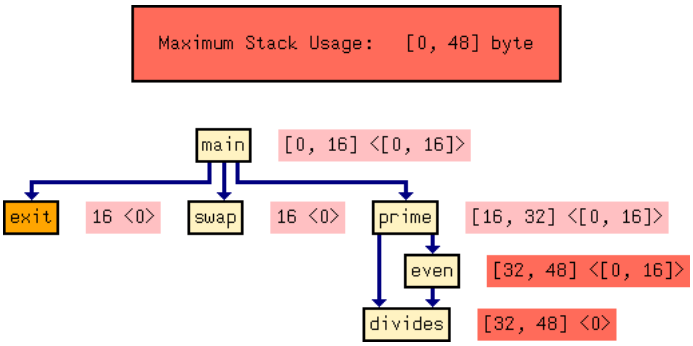

Figure 1. Call graph with stack analysis results

The results of **StackAnalyzer** are presented as annotations in a combined call graph and control-flow graph. Figure 1 shows the call graph of a small application, with stack analysis results at routines and for the entire application (at the top). On this level, the results of stack analysis are displayed in boxes located to the right of the boxes representing the routines of the application. Each result box carries two results: a *global result*, coming first, and a *local result*, following in angular brackets. Each result is an interval of possible stack levels.

The local result at a routine R indicates the stack usage in R considered on its own: It is an interval showing the possible range of stack levels within the routine, assuming value 0 at routine entry. The local result for a routine is derived from the results at individual instructions, which are shown in Figure 2 for one of the routines of this example.

The global result for routine R indicates the stack usage of R in the context of the entire application. It is an interval providing bounds for the stack level while the processor is executing instructions of R, for all call paths from the entry point to R. Thus, the global result at routine R does not include the stack usage of the routines called by R.

The predicted worst-case stack usages of individual tasks in a system can be used in an automated overall stack usage analysis for all tasks running on an Electronic Control Unit, as described in [5] for systems managed by an OSEK/VDX real-time operating system.

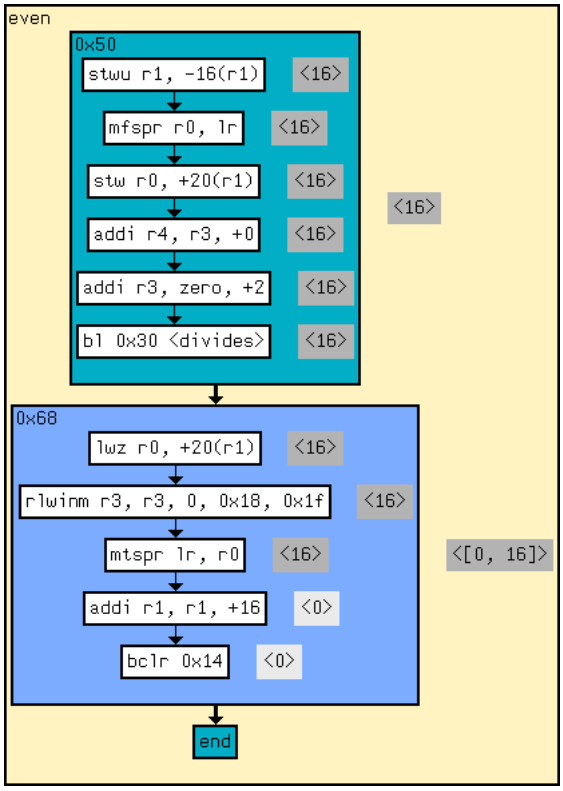

Figure 2. Individual instructions with stack analysis results

3.2 Worst-Case Execution Time

Many tasks in safety-critical embedded systems have hard real-time characteristics. Failure to meet deadlines may be as harmful as producing wrong output or failure to work at all. Yet the determination of the worst-case execution time (WCET) of a task is a difficult problem because of the characteristics of modern software and hardware [6]. Underestimating the execution time leads to systems that are prone to errors because of timing failures, whereas overestimating might lead to the wrong conclusion that the system designed will not be able to run on the selected hardware or that so much capacity is already used that no new functionality can be added.

Embedded control software (e.g., in the automotive industries) tends to be large and complex. The software in a single electronic control unit typically has to provide different kinds of functionality. It is usually developed by several people, several groups or even several different providers. It is typically combined with third-party software such as real-time operating systems and/or communication libraries.

Caches and branch target buffers are used in virtually all performance-oriented processors to reduce the number of accesses to slow memory. Pipelines enable acceleration by overlapping the executions of different instructions. Consequently the timing of the instructions depends on the execution history.

The widely used classical methods of predicting execution times are not generally applicable. Software monitoring and dual-loop benchmark modify the code, which in turn changes the cache behavior. Hardware simulation, emulation, or direct measurement with logic analyzers can only determine the execution time for some fixed inputs. They cannot be used to infer the execution times for all possible inputs in general.

In contrast to that, abstract interpretation can be used to efficiently compute a safe approximation for all possible cache and pipeline states that can occur at a program point in any program run with any input. These results can be combined with ILP (Integer Linear Programming) techniques to predict a safe upper bound of the worst-case execution time (WCET bound) and a corresponding worst-case execution path.

AbsInt's WCET tool **aiT** determines the WCET of a program task in several phases [7] (see Figure 3).

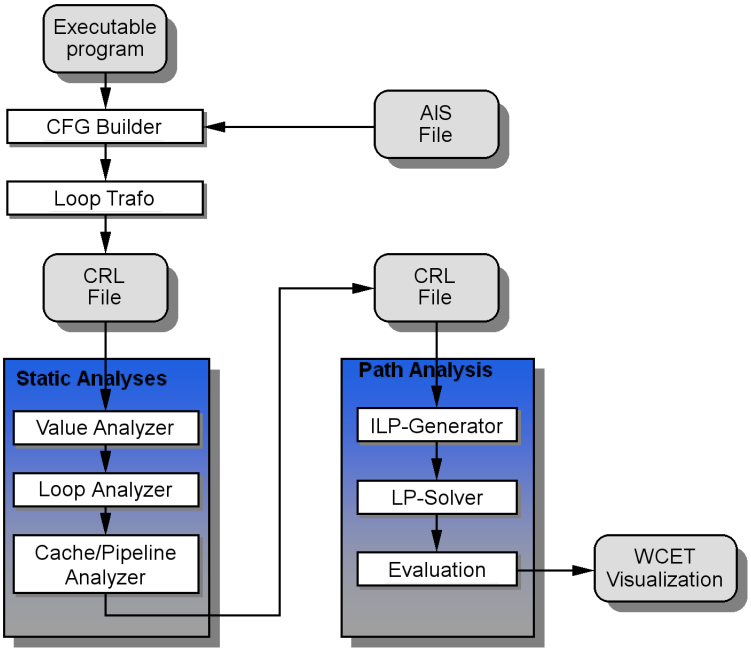

Figure 3. Phases of WCET computation

The starting point of **AbsInt**'s analysis framework is a binary program and additional information about numbers of loop iterations, upper bounds for re-

cursion, etc. This information may appear in a separate parameter file called AIS file, or as special comments in the C source that can be generated by **ASCET**.

In the first step a *decoder* reads the executable and reconstructs the control flow [8]. This requires some knowledge about the underlying hardware, e.g., which instructions represent branches or calls. The reconstructed control flow is annotated with the information needed by subsequent analyses and then translated into CRL (Control-Flow Representation Language) – a human-readable intermediate format designed to simplify analysis and optimization at the executable/assembly level. This annotated control-flow graph serves as the input for micro-architecture analysis.

Then, *value analysis* tries to determine the values in the processor registers for every program point and execution context. Its results are used in loop bound analysis and in cache analysis (possible addresses of indirect memory accesses). Value analysis can also determine that certain conditions always evaluate to true or always evaluate to false. As consequence, certain paths controlled by such conditions are never executed. Therefore, their execution time does not contribute to the overall WCET of the program, and need not be determined in the first place.

WCET analysis requires that upper bounds for the iteration numbers of all loops be known. **aiT** tries to determine the number of loop iterations by *loop bound analysis*, but succeeds in doing so for simple loops only. Bounds for the iteration numbers of the remaining loops must be provided as specifications in the AIS file or annotations in the C source.

Cache analysis classifies the accesses to main memory. The analysis in **aiT** is based upon [9], which handles analysis of caches with LRU (Least Recently Used) replacement strategy. However, it had to be modified to reflect the non-LRU replacement strategies of common microprocessors: the pseudo-round-robin replacement policy of the ColdFire MCF 5307, and the PLRU (Pseudo-LRU) strategy of the PowerPC MPC 750 and 755. The modified algorithms distinguish between sure cache hits and unclassified accesses. The deviation from perfect LRU is the reason for the reduced predictability of the cache contents in case of ColdFire 5307 and PowerPC 750/755 compared to processors with perfect LRU caches [10,11], leading to higher estimates of the WCET.

Pipeline analysis models the pipeline behavior to determine execution time bounds for sequential flows (basic blocks) of instructions as done in [12]. It takes into account the current pipeline state(s), in particular resource occupancies, contents of prefetch queues, grouping of instructions, and classification of memory references by cache analysis. The result is an execution time bound for each basic block in each distinguished execution context.

Using the results of the micro-architecture analyses, *path analysis* determines a safe upper bound of the WCET. The program's control flow is modeled by an integer linear program [13,14] so that the solution to the objective function is the predicted worst-case execution time bound for the input program. A special mapping of variable names to basic blocks in the integer linear program enables execution and traversal counts for every basic block and edge to be computed.

aiT's results are written into a report file from which they may be extracted by the **ASCET** system. In addition, **aiT** produces a graphical description that can be visualized by the **aiSee** tool [15] to view detailed information delivered by the analysis.

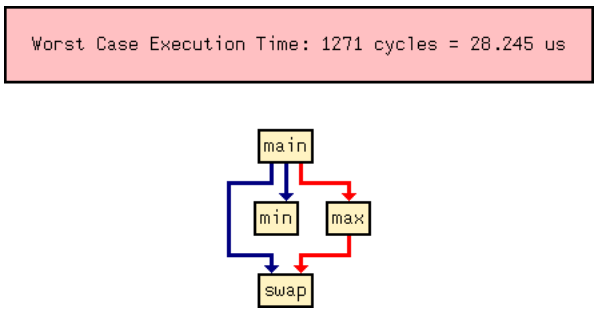

Figure 4. Call graph with WCET bounds

Figure 4 shows the graphical representation of the call graph for a small example. The calls (edges) that contribute to the worst-case runtime are marked by the color red. The computed WCET bound is given in CPU cycles and in microseconds provided that the cycle time of the processor has been specified.

Figure 5 shows the basic block graph of a loop. The number `sum #` describes the number of traversals of an edge in the worst case, while `max t` describes the execution time bound determined by **aiT** for the basic block from which the edge originates (taking into account that the basic block is left via the edge). The worst-case path, the traversal numbers and timings are determined automatically by **aiT**. Upon special command, **aiT** provides information on the origin of these timings by displaying the cache and pipeline states that may occur within a basic block.

4 Integration into the Software Development Process

When developing software with **ASCET**, reusable components are typically combined into a project. Through this project, the various operating system tasks can be configured and code generation and build can be started. Therefore, the project is the main center of the generate/compile/link-workflow. To integrate the analysis tools into this workflow, a graphical tool has been developed that takes the code and binaries created by **ASCET**, calls **aiT** and **StackAnalyzer** and displays the results in a window of its own (see Figure 6). This tool can be called directly from the project window. It fetches the information needed about the project via **ASCET**'s extensible tool API.

In addition to the information calculated by **aiT** and **StackAnalyzer**, our tool also analyzes the generated map file to calculate the total memory usage.

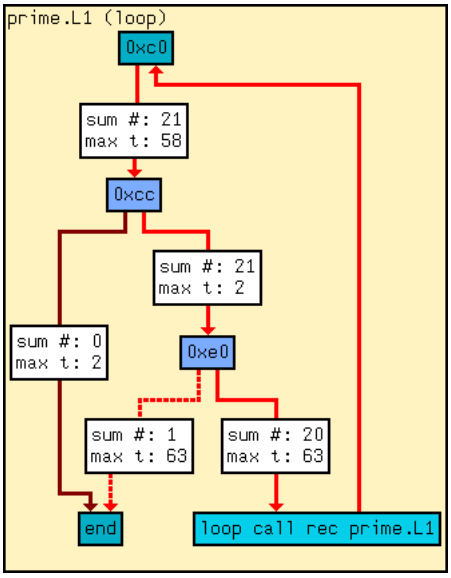

Figure 5. Basic block graph in a loop, with timing information

This tool serves as a one-stop information center that can be used to quickly review the effects of changes made to the project.

The **ASCET** software development framework is specifically suited for the development of real-time systems. The configuration of the operating system necessary for embedded development is directly built into **ASCET** projects. Processes and messages allow interfacing with real-time operating systems based on the OSEK standard. We therefore decided to base our integration on the pre-existing structure of operating system tasks and processes (called by the tasks). This way, the user can quickly check whether the actual worst-case task runtimes clash with the task scheduling periods chosen. It is planned to improve the integration by communicating the calculated WCET information back into **ASCET** to correct or improve the operating system configuration.

4.1 Improving Function Development

For the developer, the immediate and detailed feedback provided helps to find the critical areas of the project where most of the resources are spent. It also can help to decide between different alternatives to solve a given problem. Using model-based design, different modeling techniques can lead to strongly varying code. Here, the information provided by **aiT** and **StackAnalyzer** can help to prototype and develop software more rapidly.

An application of this is given in [16]. For his thesis, Abhik Dey has used our integration of **aiT** into **ASCET** to identify the components of a lambda probe model that need to be optimized for code size and performance. He was able to

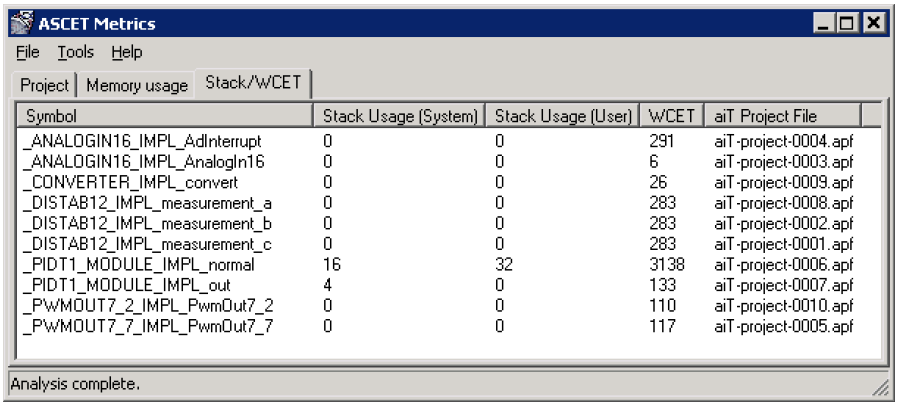

Figure 6. Window with analysis results

measure the impact of different modelling techniques as well as compiler settings precisely and managed to reduce the WCET of his complex lambda probe task down by 87% and the code size by 54%.

4.2 Improving the Complete Process

Through our tool, the project manager receives the information necessary to make choices for the project regarding modelling guidelines, code generation settings, compiler tools and compiler settings. By analyzing a set of representative projects, the complete tool chain can be optimized. As **aiT** also takes into account the exact hardware configuration and memory layout, various alternative platforms and configurations can easily be tested to achieve optimal resource usage.

Static analysis tools are also a valuable addition for managers coordinating the development of several pieces of software that will need to work together or even will be distributed on the same ECUs. By establishing memory and runtime quotas for individual parts of the software and checking and enforcing them using **ASCET**, **aiT** and **StackAnalyzer**, it is possible to prevent divergence of the efforts of several teams working on the same project.

4.3 Additional Improvements over Manual Coding

For the calculated WCET and stack usage to be useful, they need to be as close to the realistic model values as possible. Therefore, it is important to improve the precision of the calculation.

Instead of just relying on the C source code, we can make use of the additional information provided by the model, which is usually much more rigidly defined than the resulting C code. One example would be the implementation of physical values as integer values in the program: A temperature ranging from

-20.0 to +50.0 in the model might be implemented as a 16 bit integer value in the C code, with appropriate conversions. Such implementations are used even when developing using C. In this case, the developer has to take care that conversions take place in all cases where the value is used. When using model-based development coupled with a code generator, all conversions are taken care of by the code generator automatically. After specifying the range and number of digits, the developer can work with a physical view of the variable.

In this example, the implementation of the physical value does not make use of the full range of the 16 bit integer. Depending on the way that the conversions are performed, this might or might not be obvious to the static analysis tool (and to a less careful human reader of the source code). Model-based development tools are able to supply exact information on the value range used, therefore allowing for higher precision of the analysis results. Further information to be supplied could be the maximum iteration number of loops, possible values of pointers, etc. Normally, **aiT** tries to find such information by static analysis, but relies on user annotations in cases where static analysis does not succeed.

Conversely, the results of **aiT** and **StackAnalyzer** are meant to be used to improve the C code used in the project. A developer working on C code discovering a certain coding pattern to be inefficient has to change every instance of this pattern as well as monitor future changes for instances of the pattern. Using model-based development, this information can be used to improve the implementation of the code generator. To update the C code, it is enough to regenerate it using the new version of the code generator. In addition to this manual improvement process, there are a few instances where it could be possible to use the information gained by static analysis to automatically configure the code generator. One example would be in- and outlining of state machines. Here, there are few rules to choose the implementation with the best performance that apply for all state machines. Instead, the developer could generate different variants of the state machine and choose the parameter setting that results in the code performing best.

We were able to use additional information supplied by the code generator in the context of interpolation routines for characteristic tables. Here, a special loop construct is used in the C code where the loop iteration count depends on the size of the data structures involved. This special loop construct could not be resolved by static analysis, rendering the calculation of loop bounds impossible. By providing annotations on the nature of the loop bounds, we were able to calculate the WCET for the table accesses.

4.4 Experiments

The software used in the experiments was an engine throttle control module specified in **ASCET** and compiled with Tasking compiler v7.5. The compiled code was run on an STM ST10F269 microcontroller board. Run-times were extracted from bus traces made with the ISYSTEMS ILA 128 logic analyzer.

In general, finding a worst-case input for each procedure can be very challenging. In our experiments, we used worst-case path information provided by **aiT** to manually construct a corresponding input.

In order to allow a fully automatic analysis, some adaptations were necessary that are described in the following.

Volatile Variables. Some data structures in the generated code were statically initialized with the `volatile` qualifier. **aiT** uses the values from the initialized data segment of the executable for value analysis, in particular to find infeasible paths and to determine loop bounds. Since **aiT** works on the binary level on which no information about volatile variables is preserved, it requires that all volatile variables be declared as volatile by means of annotations. Without these annotations, the initializations for variables produced by **ASCET** do not lead to the worst-case path.

For example, in the following (simplified) code the variables `active` and `noOfTransfers` were both initialized to zero. Without annotations, **aiT** would consider the `true` branch of the `if` statement as infeasible and derive a loop iteration count of zero.

```
if (active) {
    ...
    dst_ptr = ...;
    adr_ptr = ...;
    end_dst_ptr = dst_ptr + noOfTransfers;
    while (dst_ptr < end_dst_ptr) {
        *dst_ptr++ = *adr_ptr++;
    }
}
```

Currently, volatile annotations must be written manually. For the future, **ASCET** is expected to pass information about volatiles to **aiT** automatically.

Synchronization. In the following example the boolean variable `_condition` is set externally by another process.

```
while (_condition) {
    ...
}
```

In such a case, an upper bound for the number of loop iterations cannot be determined statically.

This code is used to synchronize processes. Here we use an annotation specifying that this condition is never true. The cost for the synchronization should be taken into account by a system-wide schedulability analysis.

Interpolation Functions. The generated code contains lots of interpolation routines using iterative search algorithms like binary search or linear search. The loop bounds for these algorithms usually depend on some parameters, e.g., the size of the problem in case of binary search. Therefore, **aiT** has been extended by parametric loop bounds featuring an expression instead of a fixed number, for instance

```
loop here max  ceil (log2 (R4/2));
```

In this case, value analysis tries to determine the contents of register `R4` at the place of the annotation, and if successful evaluates the expression to obtain a concrete loop bound. Parametric loop bounds can thus be used to specify automatic loop bounds for **ASCET** interpolation routines without effort for **ASCET** users.

4.5 Discussion of the Results

Table 1 shows the results of practical experiments for **aiT**. The measured and analyzed times are given in processor cycles. Overall, the predicted WCET bounds are very precise.

Procedure name	**Measured**	**aiT**	**Overestimation**
_ANALOGIN16_IMPL_AdInterrupt	291	291	0.0%
_ANALOGIN16_IMPL_AnalogIn16	6	6	0.0%
_CONVERTER_IMPL_convert	26	26	0.0%
_DISTAB12_IMPL_measurement_a	263	283	7.6%
_DISTAB12_IMPL_measurement_b	263	283	7.6%
_DISTAB12_IMPL_measurement_c	263	283	7.6%
_PIDT1_MODULE_IMPL_normal	2980	3138	5.3%
_PIDT1_MODULE_IMPL_out	133	133	0.0%
_PWMOUT7_2_IMPL_PwmOut7_2	109	110	0.9%
_PWMOUT7_7_IMPL_PwmOut7_7	116	117	0.9%

Table 1. Comparison of maximal measured run-times and WCETs predicted by **aiT** (in cycles)

Table 2 compares the results of stack usage analysis with results obtained from simulator runs, showing that the analysis results are precise. All stack sizes are given in bytes. The user stack usage is 0 in most routines since the generated code rarely contains local variables that would be stored on the stack. The system stack usage is 0 in those routines that do not call other routines and therefore never push a return address on the stack.

Procedure name	Simulated	StackAnalyzer	Overestimation
_ANALOGIN16_IMPL_AdInterrupt	0/0	0/0	0.0%
_ANALOGIN16_IMPL_AnalogIn16	0/0	0/0	0.0%
_CONVERTER_IMPL_convert	0/0	0/0	0.0%
_DISTAB12_IMPL_measurement_a	0/0	0/0	0.0%
_DISTAB12_IMPL_measurement_b	0/0	0/0	0.0%
_DISTAB12_IMPL_measurement_c	0/0	0/0	0.0%
_PIDT1_MODULE_IMPL_normal	16/32	16/32	0.0%
_PIDT1_MODULE_IMPL_out	4/0	4/0	0.0%
_PWMOUT7_2_IMPL_PwmOut7_2	0/0	0/0	0.0%
_PWMOUT7_7_IMPL_PwmOut7_7	0/0	0/0	0.0%

Table 2. Comparison of maximal simulated system/user stack usage and usage as predicted by **StackAnalyzer** (in bytes)

5 Conclusion

Tools based on abstract interpretation can perform static program analysis of embedded applications. Their results hold for all program runs with arbitrary inputs. Employing static analyzers is thus orthogonal to classical testing, which yields very precise results, but only for selected program runs with specific inputs. The usage of static analyzers enables one to develop complex systems on state-of-the-art hardware, increases safety, and saves development time. Precise stack usage and timing predictions enable the most cost-efficient hardware to be chosen. As recent trends in the automotive industry (e.g., X-by-wire, time-triggered protocols) require knowledge of the WCETs of tasks, a tool like **aiT** is of high importance.

Combined with model-based design and automatic code generation, the potential of static analysis tools is increased greatly: More strict specification and development guidelines enforced by tools like **ASCET** allow for a high precision of the analyzers' estimates as demonstrated by our experiments. The resulting combination allows for the development of more secure and better-performing systems while decreasing time-to-market through enhancing development productivity.

Since memory class information is only finalized in the linking stage, **aiT** and **StackAnalyzer** currently operate on completely linked binaries. This is not always convenient for the user. Many companies rely on a complicated toolchain to create binaries for the embedded platforms. It would be a huge overhead to use this toolchain just to analyze the performance of a single component. We currently research different ways of analyzing single compiled object files, either through direct analysis of the object file or through linking just the object file, ignoring or providing undefined symbols. In both cases, the calculated WCET bounds will be higher (and therefore less exact) due to the information missing from the linking stage. But even those less exact results might help users to improve the performance of their component.

We plan to further improve on the solution that we have developed so far by integrating static analysis tools like **aiT** and **StackAnalyzer** even more tightly into **ASCET**'s development environment. By allowing developers to analyze smaller parts of a model without integrating them into a project, we would be able to decrease turn-around-times for function development even more. We also hope to use the results obtained by **aiT** for semi-automatic OS configuration of the whole project.

Acknowledgments

The collaboration between AbsInt GmbH and Universität des Saarlandes was supported by the Network of Excellence on Embedded Systems Design **ARTIST2**. Collaboration between AbsInt GmbH and ETAS GmbH has been partially supported by the FP6 STREP project **INTEREST** (INTEgrating euRopean Embedded Systems Tools).

References

1. The Motor Industry Software Reliability Association: Guidelines for the Use of the C Language in Critical Systems. (2004) ISBN 0-9524156-2-3.
2. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution time problem - overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems **5** (2007) 1–47
3. Schätz, B., Hain, T., Prenninger, W., Rappl, M., Romberg, J., Slotosch, O., Strecker, M., Wisspeintner, A., et al.: CASE tools for embedded systems. Technical Report TUMI-0309, Fakultät für Informatik, TU München (2003)
4. Damm, W., Schulte, C., Wittke, H., Segelken, M., Higgen, U., Eckrich, M.: Formale Verifikation von ASCET Modellen im Rahmen der Entwicklung der Aktivlenkung. In: INFORMATIK 2003 – Innovative Informatikanwendungen. Volume 34 of Lecture Notes in Informatics. (2003) 340–344
5. Janz, W.: Das OSEK Echtzeitbetriebssystem, Stackverwaltung und statische Stackbedarfsanalyse. In: Embedded World, Nuremberg, Germany (2003)
6. Wilhelm, R.: Determining bounds on execution times. In Zurawski, R., ed.: Handbook on Embedded Systems. CRC Press (2005) 14–1 – 14–23
7. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: Proceedings of EMSOFT 2001, First Workshop on Embedded Software. Volume 2211 of Lecture Notes in Computer Science., Springer-Verlag (2001) 469–485
8. Theiling, H.: Extracting Safe and Precise Control Flow from Binaries. In: Proceedings of the 7th Conference on Real-Time Computing Systems and Applications, Cheju Island, South Korea (2000)
9. Ferdinand, C.: Cache Behavior Prediction for Real-Time Systems. PhD thesis, Saarland University (1997)

10. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The influence of processor architecture on the design and the results of WCET tools. Proceedings of the IEEE **91**(7) (2003) 1038–1054 Special Issue on Real-Time Systems.
11. Reineke, J., Grund, D., Berg, C., Wilhelm, R.: Predictability of cache replacement policies. Reports of SFB/TR 14 AVACS 9, SFB/TR 14 AVACS (2006) ISSN: 1860-9821, http://www.avacs.org.
12. Schneider, J., Ferdinand, C.: Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems. Volume 34. (1999) 35–44
13. Theiling, H., Ferdinand, C.: Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In: Proceedings of the 19th IEEE Real-Time Systems Symposium, Madrid, Spain (1998) 144–153
14. Theiling, H.: ILP-based interprocedural path analysis. In Sangiovanni-Vincentelli, A.L., Sifakis, J., eds.: Proceedings of EMSOFT 2002, Second International Conference on Embedded Software. Volume 2491 of Lecture Notes in Computer Science., Springer-Verlag (2002) 349–363
15. AbsInt Angewandte Informatik GmbH: aiSee Home Page. http://www.aisee.com. (2006)
16. Dey, A.: Implementation of control algorithms in production code projects, using case tools with automated code generation. Master's thesis, FHT Esslingen (2006)