Verifying non-functional real-time properties by static analysis

Christian Ferdinand, Reinhold Heckmann

Session 7B: Verification & Validation

Title: Verifying non-functional real-time properties by static analysis

Authors:

Christian FERDINAND (speaker)
AbsInt Angewandte Informatik GmbH
e-mail : ferdinand@absint.com
phone : +49 681 831 831 7

Reinhold Heckmann (co-author)
AbsInt Angewandte Informatik GmbH
e-mail : heckmann@absint.com
phone : +49 681 831 830 3

Jean SOUYRIS (co-author)
AIRBUS France
e-mail : jean.souyris@airbus.com
phone : +33 561 181 261

Keywords: Safety, Estimation of Stack Usage, Timing Validation, Schedulability Analysis, WCET (worst-case execution time) Prediction

Introduction

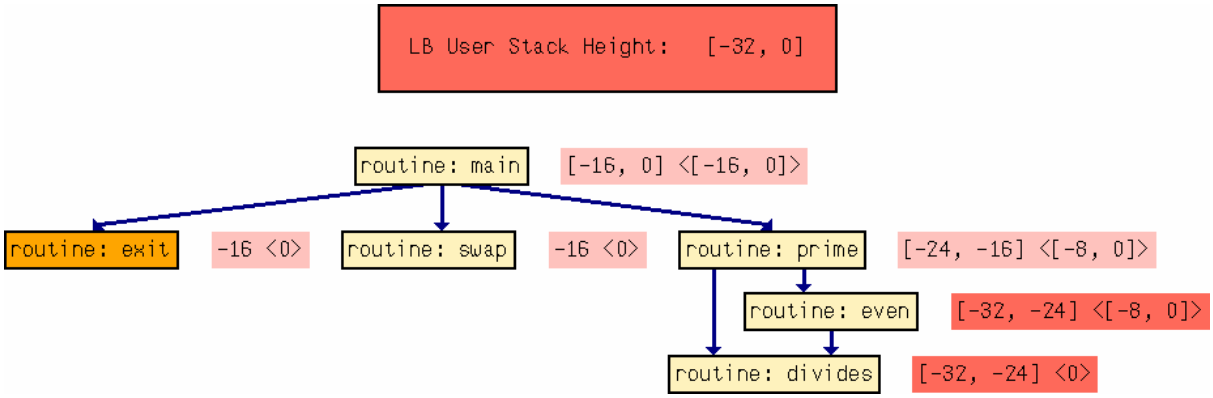
Static analyzers based on *abstract interpretation* are tools aiming at the automatic detection of run-time properties by analyzing the source, assembly or binary code of a program. From Airbus' point of view, the first interesting properties covered by static analyzers available on the market, or as prototypes coming from research, are absence of run-time errors, maximum stack usage and Worst-Case Execution Time (WCET). This paper will focus on the two latter.

Stack analyzers

The failure of a safety-critical application on an embedded processor may lead to severe damage. A possible cause of catastrophic failure is *stack overflow* that usually leads to run-time errors that are difficult to diagnose. The problem is that the memory area for the stack must be reserved by the programmer. Underestimation of the maximum stack usage leads to stack overflow, while overestimation means wasting memory resources. Measuring the maximum stack usage with a debugger is no solution since one only obtains a result for a single program run with fixed input. Even repeated measurements with various inputs cannot guarantee that the maximum stack usage is ever observed. Some, but not all compilers provide information about stack usage, but this requires the availability of the source code, and the information becomes invalid when the generated code is optimized by hand or by some automatic tool.

The main job of stack analyzers is to compute an upper bound of the maximum stack consumption of a program by statically analyzing its binary. The way this upper bound is computed makes it both safe and precise, thanks to the theory of abstract interpretation. By "safe", it is understood that the analysis result really is an upper bound of the maximum stack depth for all possible executions of the program; by "precise", we mean: the maximum stack consumption computed by the tool is lower than the one computed by classical and usually very conservative methods.

Stacks are used to store return addresses, function parameters, local variables, and intermediate results. Since it is difficult to predict on source-code level the amount of data stored on the stack by the compiler, **StackAnalyzer** operates on the generated assembler files or executables that contain the necessary information. Therefore the tool must be adapted to the microprocessor the analyzed program is compiled/assembled for. For covering most of its needs, Airbus is currently validating AbsInt's stack analyzers for the X86, Motorola PowerPC, and Texas TMS320C3x families. In addition, AbsInt offers stack analyzers for C166/ST10, Arm7 TDMI, and HC12 processors.



The picture shows the call graph of a small example application for PowerPC 755. The red box at the top indicates the stack usage of the entire program. The yellow and orange boxes stand for routines, and the arrows in-between indicate routine calls. The stack analysis results for the routines appear in the boxes to the right of the routine names. Each result box carries two results: a global result, coming first, and a local result, following in angular brackets. For instance, routine prime has global result [–24, –16] and local result [–8, 0]. Each result is an interval of possible stack pointer values. Intervals of the form [*n*, *n*] are abbreviated to *n*. The result node for a routine is displayed in red if the lower end of its global result interval equals the lower end of the interval for the entire program. All other result nodes at routines are displayed in pink.

Since the stack grows by decreasing the stack pointer, all the results in the picture are negative. 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 pointer values within the routine, assuming value 0 at routine entry. The global result for routine *R* indicates the stack usage of *R* in the context of the whole application. It is an interval providing bounds for the values of the stack pointer 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*. For instance, routine prime has a global result of [–24, –16] because the two calls of prime in main have stack level –16, and prime itself has local stack usage [–8, 0]. Routine divides has a global result of [–32, –24] because its local stack usage is 0 and it is called in prime at the accumulated stack level (–16)+(–8) = –24 and in even at the accumulated stack level (–16)+(–8)+(–8) = –32.

Upon request, **StackAnalyzer** shows the control-flow graphs of the routines at basic-block level or instruction level, annotated with local stack analysis results. The results at instructions are determined by assuming a stack level of 0 at routine entry, and then taking the effects of each instruction into account. The result at a basic block is the join of the results for its instructions. Similarly, the local result at a routine is the join of the results for its basic blocks and the value 0 assumed at routine entry. Here, the join of a collection of intervals and numbers is the smallest interval containing all these intervals and numbers.

StackAnalyzer is a tool useful for developers of applications for embedded systems. Used properly, it reduced the production effort and avoids run-time errors due to stack overflow. Critical program parts are easily identified by the color of the result boxes. This gives useful hints for optimizing the stack usage of the application.

`StackAnalyzer` can also be used to guarantee the absence of stack overflow for quality control and certification of software.

Thanks to AbsInt's stack analysers, at least 10 avionics applications safely compute the amount of memory they reserve for their stacks. A significant example is a six year old x86 application in which the maximum stack usages - one per thread - were computed by monitoring the stacks at test time and then by doubling the maximum value observed for each stack. In spite of doubling the measured values, it is not possible to prove that an upper bound of the stack consumption has been determined in this way. Now, with AbsInt's stack analyzer, it can be stated that the upper bounds stack for the stack consumption computed by the tool - one per thread stack - are safe and, as they are all close to (from above: they are safe!) the measured ones before doubling, almost half of the memory previously dedicated to the stacks is now available for another usage.

Worst-Case Execution Time analyzers (`aiT` family)

Timing Validation

Computers controlling potentially hazardous machinery are expected to always execute in time. Consequently, for the modelling and planning of embedded systems it is essential that their timing behavior be validated. For several reasons timing validation is challenging:

- The increasing performance of the microcontrollers allows to put more and more functionality on a single embedded control unit. This means that there is a trend to more and more tasks in a real-time system with complicated dependencies and scheduling requirements and multiple levels of interrupts. The complex interaction of different functions on one microcontroller is often managed by a real-time operating system (RTOS). It is typically not possible nor practical to test the system with all possible inputs and all distributions over time of "external events" that might lead to an interrupt.
- Software is typically written in some high level language like C, C++, Java, or Ada. Optimizing compilers make it difficult for the developers to precisely estimate the execution time of their code.
- The software is typically developed in teams. Furthermore, the share of 3rd party software like RTOS and communication libraries is increasing. The timing behavior of the interacting software components is rarely known.
- Modern processor components like caches and pipelines complicate the task of determining the WCET considerably, since the execution time of a single instruction may depend on the execution history. Real-time systems are typically composed of a set of tasks with specified deadlines (mostly dictated by the surrounding physical environment).

A schedulability analysis has to be performed in order to guarantee that all timing constraints will be met (*timing validation*) [15]. This requires that the system be designed in a way that schedulability analysis is possible. There exist many results and analysis methods for real-time scheduling, see e.g. [11, 14, 23, 12, 17, 7, 2, 16, 6].

Another approach to design (complex) safety critical real-time systems is the time-triggered approach in contrast to the event-triggered approach. In a time-triggered system all actions are performed at times that have been fixed during system design, i.e., a static schedule for all tasks is determined before the system runs. The time-triggered approach has successfully been used e.g. to implement fly-by-wire systems for aircrafts. In combination with time-triggered communication busses (TTA, FlexRay, TTCAN, SAFEbus, ...) and time-triggered operating systems (e.g. OSEKtime) it is possible to design safety-critical distributed real-time systems. The time-triggered approach is recently becoming more used in the automotive area, e.g. for X-by-wire applications.

Schedulability analysis for event-triggered systems and determining a static schedule for time-triggered systems require the worst-case execution time of each task in the system to be known prior to its execution. Since this is not computable in general, estimates of the WCET have to be calculated. These estimates have to be safe, i.e., they must never underestimate the real execution time. Furthermore, they should be tight, i.e., the overestimate should be as small as possible.

Worst Case Execution Time Prediction

There is typically a large gap between the cycle times of modern microprocessors and the access times of main memory. Caches and branch target buffers are used to overcome this gap in virtually all performance-oriented processors (including high-performance microcontrollers and DSPs). Pipelines enable acceleration by overlapping the executions of different instructions. The consequence is that the execution behavior of the instructions cannot be analysed separately since this depends on the execution history.

Cache memories usually work very well, but under some circumstances minimal changes in the program code or program input may lead to dramatic changes in cache behavior. For (hard) real-time systems, this is undesirable and possibly even hazardous. The widely used classical methods of predicting execution times are not generally applicable. Software monitoring or the dual loop benchmark change the code, what in turn has impact on the cache behavior. Hardware simulation, emulation, or direct measurement with logic analyzers can only determine the execution time for one input. This cannot be used to infer the execution times for all possible inputs in general. Making the safe—yet for the most part—unrealistic assumption that all memory references lead to cache misses results in the execution time being overestimated by several hundred percent.

WCET Computation

In our approach [5] the determination of the WCET of a program task is composed of several phases:

Value Analysis: computation of address ranges for instructions accessing memory.

Cache Analysis: classification of memory references as cache misses or hits [4].

Pipeline Analysis: prediction of the behavior of the program on the processor pipeline [10].

Path Analysis: the determination of a worst-case execution path of the program [21].

The results of the value analysis are used by the cache analysis to predict the behavior of the (data) cache. The results of cache analysis are used within pipeline analysis allowing the prediction of pipeline stalls due to cache misses. The combined results of the cache and pipeline analyses are used to compute the execution times of program paths. The separation of WCET determination into several phases has the additional effect that different methods tailored to the subtasks can be used. Value analysis, cache analysis, and pipeline analysis are done by *abstract interpretation* [3], a semantics-based method for static program analysis. Integer linear programming is used for path analysis.

Reconstruction of the Control Flow from Binary Programs

The starting point of our analysis framework is a binary program and additional user-provided information about numbers of loop iterations, upper bounds for recursion, etc. In the first step a parser reads the compiler output and reconstructs the control flow [18, 19]. 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). This annotated control flow graph serves as the input for micro-architecture analysis.

Pipeline Analysis

Pipeline analysis models the pipeline behavior to determine execution times for a sequential flow (basic block) of instructions, as done in [10, 13]. 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 as cache hits or misses. The result is an execution time for each instruction in each distinguished execution context.

Path Analysis

Using the results of the micro-architecture analyses, path analysis determines a safe estimate of the WCET. The program's control-flow is modelled by an integer linear program [21, 20] so that the solution to the objective

function is the predicted worst-case execution time 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.

Analysis of Loops and Recursive Procedures

Loops and recursive procedures are of special interest since programs spend most of their runtime there. Treating them naively when analysing programs for their cache and pipeline behavior will result in a high loss of precision.

The following observation can be made frequently: the first execution of the loop body usually loads the cache and subsequent executions find most of their referenced memory blocks in the cache. Hence, the first iteration of the loop often encounters cache contents quite different from that of later iterations. This has to be taken into account when analysing the behavior of a loop on the cache. A naive analysis would combine the abstract cache states from the entry to the loop and from the return from the loop body, thereby losing most of the contents. Therefore, it is useful to distinguish the first iteration of loops from the others.

A method has been designed and implemented in the program analyzer generator **PAG** [1], which virtually unrolls loops, the so-called VIVU approach. Memory references are now considered in different execution contexts, essentially nestings of first and non-first iterations of loops.

User annotations

Apart from the executable, **aiT** needs user input to find a result at all, or to improve the precision of the result. The most important user annotations specify the targets of computed calls and branches and the maximum iteration counts of loops (there are many other possible annotations). Originally, program points had to be identified by their address in these annotations. This is cumbersome and error-prone since addresses may change after recompilation. Now a more high-level specification language was introduced for referring to program points symbolically (e.g. the second loop in routine *R*) or via source code annotations.

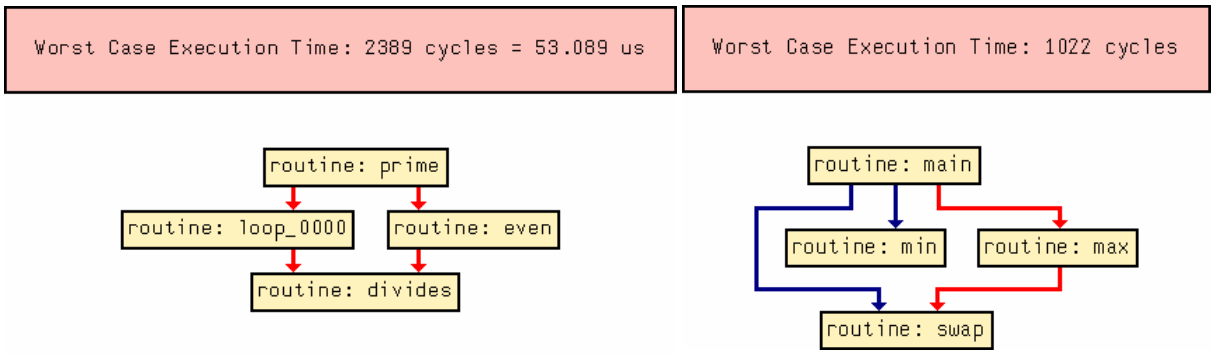
aiT – WCET Analyzers

The techniques described above have been incorporated into AbsInt's **aiT** WCET analyzer tools. They get as input:

- an executable (in ELF format). The code is generated with the Diab Data C compiler from a restricted subset of ANSI-C (no dynamic data structures, no setjmp/longjmp),
- user annotations, giving the call targets for indirect function calls and upper bounds on the iteration counts of all loops,
- a description of the (external) memories and buses (i.e. a list of memory areas with minimal and maximal access times), and
- a task (identified by a start address). A task denotes a sequentially executed piece of code (no threads, no parallelism, and no waiting for external events). This should not be confused with a task in an operating system that might include code for synchronization or communication.

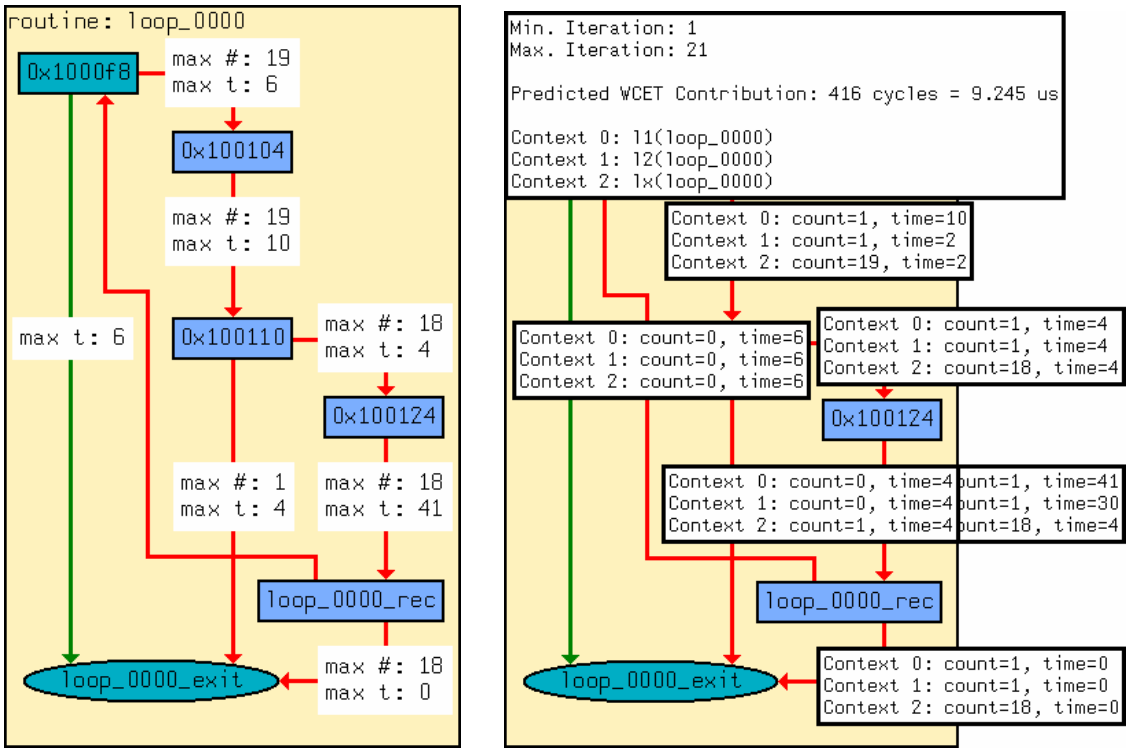
The WCET analyzers compute an upper bound of the runtime of the task (assuming no interference from the outside). Effects of interrupts, IO and timer (co-)processors are not reflected in the predicted runtime and have to be considered separately (e.g. by a quantitative analysis).

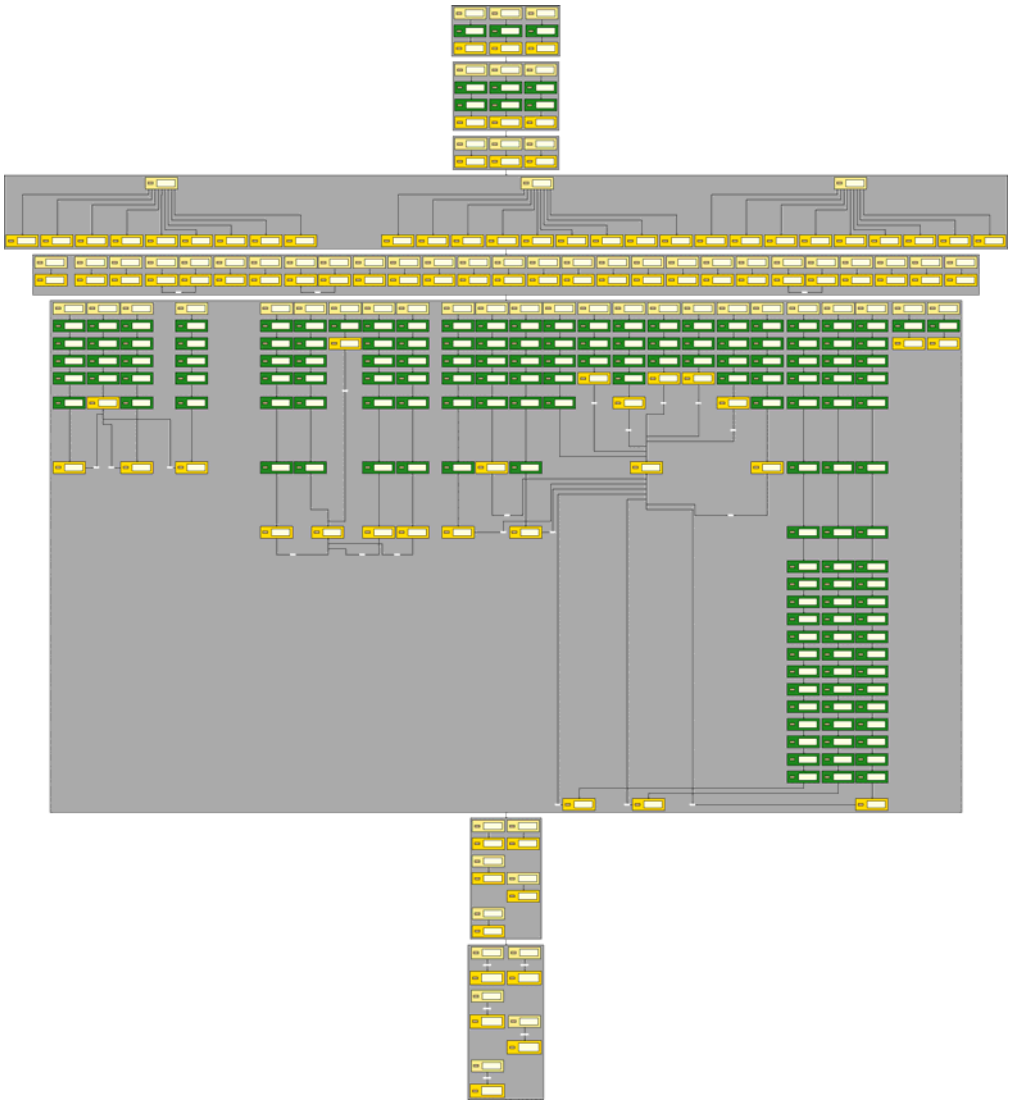
In addition to the raw information about the WCET, several aspects can be visualised by the **aiSee** tool [8] to view detailed information delivered by the analysis.



The pictures above show the graphical representation of the call graph for two small examples. The calls (edges) that contribute to the worst-case runtime are marked by the color red. The computed WCET is given in CPU cycles and in microseconds provided that the cycle time of the processor has been specified (as in the left picture).

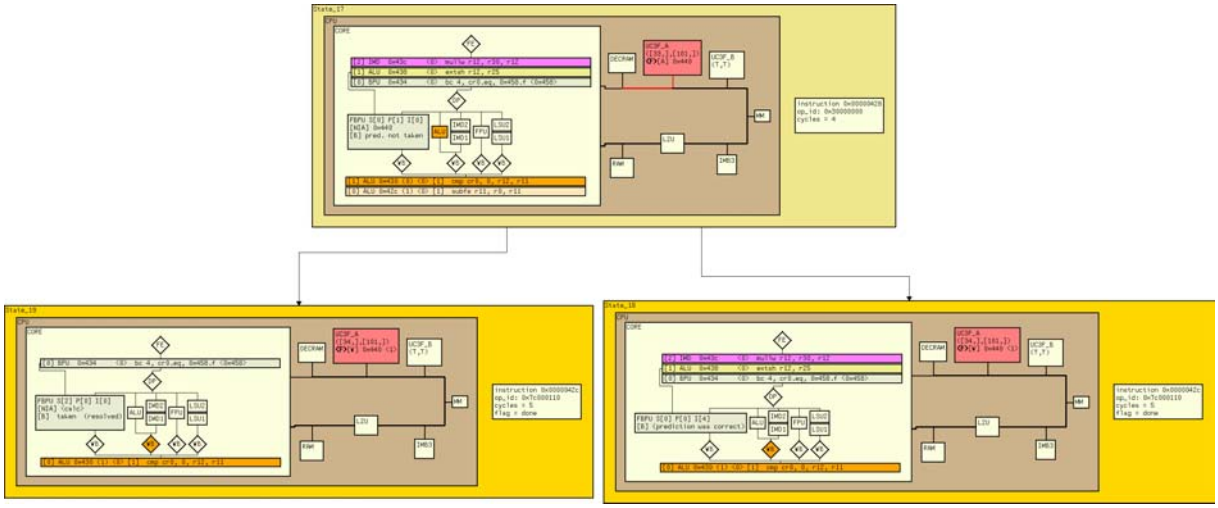
Of the two pictures below, the left one shows the basic block graph of a loop. max # describes the maximal number of traversals of an edge in the worst case. max t describes the maximal execution time of the basic block from which the edge originates (taking into account that the basic block is left via the edge). The worst-case path, the iteration numbers and timings are determined automatically by **aiT**. The right picture shows the contribution of the various execution contexts of the loop. In this example, the user has asked **aiT** to distinguish between the first iteration (Context 0), the second iteration (Context 1), and all other iterations (Context 2).





The large picture above shows the possible pipeline states for a basic block in this example. Such pictures are shown by **aiT** upon special demand. The grey boxes correspond to the instructions of the basic block, and the yellow rectangles are individual pipeline states. Their cyclewise evolution is indicated by the strokes connecting them. Each layer in the trees corresponds to one CPU cycle. Branches in the trees are caused by conditions that could not be statically evaluated, e.g. a memory access that could not be classified as a guaranteed cache hit or guaranteed cache miss.

The picture below shows some part of the picture above in greater magnification. You can see three individual pipeline states, each displaying a diagram of the architecture of the CPU (in this case a PowerPC 555) showing the occupancy of the various pipeline states with the instructions currently being executed.



Evaluation of aiT by Airbus

The main job of **aiT** is to compute an upper bound for the Worst-Case Execution Time of a program by analysing its binary. Like for stack analyzers, the way this upper bound is computed makes it both safe and precise. Again, **aiT** has to be instantiated for each microprocessor for allowing it to analyze binaries produced for a particular CPU, but also for taking into account the inner behavior of the CPU when it executes the instructions of the analyzed program. For modern architecture, modelling the cache and complex pipelines (superscalar in the PowerPC 755) and bus interfaces is the only way for computing a safe and precise WCET.

For covering most of its needs, Airbus is currently validating **aiT** tools for the following targets: Motorola ColdFire 5307, Motorola PowerPC 755, and Texas TMS320C33. (**aiT** is also available for Arm7, HCS12, and PowerPC 555.) Basically the principles that allow **aiT** to compute a safe and precise upper bound of the Worst-Case Execution Time are the following:

1. Abstract Interpretation makes it possible to compute a safe upper bound of the WCET for all possible executions (i.e., for all possible inputs of the program);
2. The Abstract Domains on which the analysis is designed allows **aiT** to compute a WCET close to the real one (but still greater than...);
3. The inner working of the microprocessor is conservatively (for safety) and precisely modeled, again by Abstract Interpretation;
4. The restrictions the analysed program must conform to are clearly identified at tool's design time and also clearly stated in the user's manual.

Airbus' first experience in AbsInt's WCET technology comes from the experimental usage of **aiT** ColdFire 5307 on a 340 flight control program for which the traditional WCET computation method was already applied. So, it was possible to compare both techniques and to conclude that by using **aiT**, once validated, one could allow to safely free about 10% of CPU power, due to the tighter WCET computed by **aiT**. The following table compares the estimated WCETs produced with Airbus's traditional method and **aiT**'s results.

Task	Airbus' method	aiT's results	precision improvement
1	6.11 ms	5.50 ms	10.0 %
2	6.29 ms	5.53 ms	12.0 %
3	6.07 ms	5.48 ms	9.7 %
4	5.98 ms	5.61 ms	6.2 %
5	6.05 ms	5.54 ms	8.4 %
6	6.29 ms	5.49 ms	12.7 %
7	6.10 ms	5.35 ms	12.3 %
8	5.99 ms	5.49 ms	8.3 %
9	6.09 ms	5.45 ms	10.5 %
10	6.12 ms	5.39 ms	11.9 %
11	6.00 ms	5.19 ms	13.5 %
12	5.97 ms	5.40 ms	9.5 %

The code analyzed didn't have to be instrumented in order to apply the tool. No changes in the development process of the programs to be analyzed were necessary.

The results of the evaluation are very encouraging. We believe that the WCET tool not only can be used in verifying that WCET constraints are met but also in earlier stages of the development process as well. At a stage when the software is already available, but working hardware is not, the tool can be used for a performance evaluation. Based on the contributions of the program parts to the WCET one can make design decisions, e.g., with respect to static scheduling or code/data placement. The effects on the cache and pipeline can be viewed using the visualization options of the tool and causes for unexpected local timing behavior identified.

Currently, AbsInt is adapting its **aiT** for PPC 755 to the timings of a proprietary chipset. Consequently, the results already obtained are based on an inappropriate model of these timings. Nevertheless, the first experiments made on benchmark code made Airbus choose **aiT** as the basis of the WCET demonstration for some critical A380 avionics computers.

Common considerations about both families of tools

None of the tools mentioned above would be as efficient as they are, or simply would not exist at all, if they were not based on the theory of Abstract Interpretation. Indeed, this theory makes them to be sound (i.e., producing safe results) at least in their design.

The soundness of these tools is also based on the hardware and sometimes software models they rely on. If these models are correct, the soundness of Abstract Interpretation fully applies. A special issue is therefore for the user to get sure that the analyzed programs conform to the set of restrictions (see user's manual), and to validate the hardware model; it must be noticed that AbsInt supports the user for this validation at both tool and methodological levels.

Biographical data for the authors

Christian FERDINAND (AbsInt): Born 1965, Ferdinand studied 1985-1990 computer science and electrical engineering at Saarland University. 1991-1998 he was academic research staff member at the chair for compiler construction and programming languages at Saarland University and participated in the ESPRIT projects PROSPECTRA (PROgramming by SPECification and TRAnsformation) and COMPARE (COMPiler generation for PARallEl machines) and in the Sonderforschungsbereich 124 (VLSI und Parallelität). In 1997, he received his Ph.D on "Cache Behavior Prediction for Real-Time Systems". C. Ferdinand received a student scholarship and a Ph.D.-student scholarship of the Siemens AG and received the Dr. Eduard Martin Preis in 1999 (award for best

Ph.D. Thesis in computer science at Saarland University). He is a co-founder of AbsInt Angewandte Informatik GmbH and since February 1998 managing director of AbsInt. His work is focused on code generation for digital signal processors and on timing analysis for real-time systems.

Reinhold HECKMANN (AbsInt) studied Computer Science at the University of the Saarland in Saarbrücken, where he received the Dr. rer. nat. degree in 1991. After being Lecturing Assistant at the University of the Saarland and Research Fellow at Imperial College, London, Reinhold Heckmann became Senior Researcher at AbsInt Angewandte Informatik GmbH. His major research areas include programming languages and compiler construction, document processing, semantics of programming languages and domain theory, exact real arithmetic, and static analysis of real-time systems, in particular cache and pipeline analysis.

Jean SOUYRIS (Airbus) is a member of the Software Verification Group of Airbus France and in charge of the research activities for the application of formal program static analysis. Research efforts conducted for more than five years yield first partial but industrially applicable results. Integration of the early tools is now effective in the verification framework used for on-going aircraft developments (safety-critical software). On-going research projects are based on abstract interpretation where efforts target the extension of the early results. As first education, he holds an engineer degree from the INSA engineer school.

References

[1] M. Alt and F. Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *Proceedings of SAS'95, StaticAnalysis Symposium*, volume 983 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[2] N. C. Audsley, A. Burns, K. W. Tindell, M. F. Richardson, and A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-Emptive Scheduling. *IEEE Transactions on Software Engineering*, 8(5), 1993.

[3] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.

[4] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Universität des Saarlandes, 1997.

[5] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R.Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, 2001.

[6] R. Gerber, S. Hoo, and M. Saksena. Guaranteeing Real-Time Requirements with Resource-Based Calibration of Periodic Processes. *IEEE Transactions on Software Engineering*, 21(7), 1995.

[7] W. A. Halang and K. M. Sacha. *Real-Time Systems*. World Scientific, 1992.

[8] http://www.aisee.com. *aiSee Home Page*.

[9] D. Kästner and S. Thesing. Cache Sensitive Pre-Runtime Scheduling. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Montreal, Canada, 1998.

[10] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline Modeling for Timing Analysis. *Proceedings of the 9th International Static Analysis Symposium*, 2002.

[11] C. L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *J. ACM*, 20(1), 1973.

[12] K. Ramamritham, J. A. Stankovic, and P.-F. Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), 1990.

[13] J. Schneider and C. Ferdinand. 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, pages 35–44, May 1999.

[14] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9), 1990.

[15] J. A. Stankovic. *Real-Time and Embedded Systems*. ACM 50th Anniversary Report on Real-Time Computing Research, 1996. http://www-ccs.cs.umass.edu/sdcr/rt.ps.

[16] J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, 28(6), 1995.

[17] A. D. Stoyenko, V. C. Hamacher, and R. C. Holt. Analyzing Hard-Real-Time Programs For Guaranteed Schedulability. *IEEE Transactions on Software Engineering*, 17(8), 1991.

[18] H. Theiling. 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.

[19] H. Theiling. Generating Decision Trees for Decoding Binaries. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Snowbird, Utah, USA, June 2001.

[20] H. Theiling. ILP-based Interprocedural Path Analysis. In *Proceedings of the Workshop on Embedded Software*, Grenoble, France, October 2002.

[21] H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, 1998.

[22] R. Wilhelm and D. Maurer. *Compiler Design*. International Computer Science Series. Addison–Wesley, 1995. Second Printing.

[23] J. Xu and D. L. Parnas. Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations. *IEEE Transactions on Software Engineering*, 16(3), 1990.