Reliable and Precise WCET and Stack Size Determination for a Real-life Embedded Application

Philippe Baufreton*, Reinhold Heckmann**

*Hispano-Suiza, Etablissement de Réau BP 42
F-77551 Moissy-Cramayel Cédex, France
philippe.baufreton@hispano-suiza-sa.com
http://www.hispano-suiza-sa.com/
**AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
heckmann@absint.com
http://www.absint.com/

Abstract. Failure of a safety-critical application on an embedded processor can lead to severe damage or even loss of life. Here we are concerned with two kinds of failure: stack overflow, which usually leads to run-time errors that are difficult to diagnose, and failure to meet deadlines, which is catastrophic for systems with hard real-time characteristics. Classical software validation methods like simulation and testing with debugging require a lot of effort, are expensive, and do not really help in proving the absence of such errors.

AbsInt's tools **StackAnalyzer** and **aiT** (timing analyzer) provide a solution to these problems. They use abstract interpretation as a formal method that leads to statements valid for all program runs. Both tools have been used successfully at **Hispano-Suiza** to analyze applications running on a Motorola PowerPC MPC555. They turned out to be well-suited for analyzing large safety-critical applications developed at **Hispano-Suiza**. They can be used either during the development phase providing information about stack usage and runtime behavior well in advance of any run of the analyzed application, or during the validation phase for acceptance tests prior to the certification review.

1 Introduction

Failure of a safety-critical application on an embedded processor can lead to severe damage or even loss of life. Therefore, utmost carefulness and state-of-the-art machinery have to be applied to make sure that an application meets all requirements. Classical software validation methods like simulation and testing with debugging require a lot of effort and are very expensive. Furthermore, they cannot really guarantee the absence of errors. In contrast, *abstract interpretation* (Cousot and Cousot, 1977) is a formal verification method that yields statements valid for all program runs with all inputs, e.g., absence of violations of timing or space constraints, or absence of runtime errors.

Nowadays tools based on abstract interpretation are commercially available and have proved their usability in industrial practice. For example, stack overflow can be detected by **AbsInt**'s **StackAnalyzer**, and violations of timing constraints are found by **AbsInt**'s **aiT** tool (Ferdinand et al., 2001) that determines upper bounds for the worst-case execution times of the tasks of an application. These tools have been successfully used at **Hispano-Suiza** to analyze large safety-critical avionics applications running on Motorola PowerPC MPC555.

Section 2 introduces value analysis, section 3 describes stack analysis with the tool **StackAnalyzer**, and section 4 presents the WCET analyzer **aiT**. Section 5 describes **Hispano-Suiza**'s application that was analyzed by **StackAnalyzer** and **aiT**, section 6 presents the results of applying **StackAnalyzer** to it, and in section 7, the results obtained from **aiT** are compared with the results of measurements. Section 8 concludes.

2 Value Analysis

Among other things, **StackAnalyzer** and **aiT** perform a *value analysis* to determine the values stored in the processor's memory for every program point and execution context. Value analysis is a static analysis method based on abstract interpretation. It produces results valid for every program run and all inputs. Therefore, it cannot always predict an exact value for a memory location, but determines *abstract values* that stand for sets of concrete values.

There are several variants of value analysis depending on what kinds of abstract values are used. In *constant propagation*, an abstract value is either a single concrete value or the statement that no information about the value is known. In *interval analysis*, abstract values are intervals that are guaranteed to contain the exact values. Value analysis in **AbsInt**'s tools is currently based on interval analysis, but extensions are being studied that would record known equalities between otherwise unknown values, or more generally, upper and lower bounds for their differences, or even more generally, arbitrary linear constraints between values.

Value analysis, even in its simple form as interval analysis, has various applications as an auxiliary method providing input for other analysis tasks including stack and WCET analysis, which are presented in the next few sections.

3 Stack Usage Analysis

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 usually 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 single program runs with fixed inputs. Even repeated measurements cannot guarantee that the maximum stack usage is ever observed.

AbsInt's tool **StackAnalyzer** provides a solution to this problem: By extracting the value of the stack pointer from value analysis, the tool can figure out how the stack increases and decreases along the various control-flow paths. This information can be used to derive an upper bound for the maximum stack usage of the entire application provided that the call depths of

recursive routines are specified by user annotations. An overestimation of the maximum stack usage may be caused by unreachable code.

The results of **StackAnalyzer** are documented in a report file and presented as annotations in a combined call graph and control-flow graph with stack analysis results at routines and for the entire application. Each routine has a local result and a global result. 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 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.

StackAnalyzer provides automatic tool support to calculate precise information on the stack usage. This not only reduces development effort, but also helps to prevent runtime errors due to stack overflow. Critical program sections are easily recognized thanks to color coding. The analysis results thus provide valuable feedback in optimizing the stack usage of an application. 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 one Electronic Control Unit, as described in Janz (2003) for systems managed by an OSEK/VDX real-time operating system.

4 WCET Analysis: Worst-Case Execution Time Prediction

Many tasks in safety-critical embedded systems have hard real-time characteristics. It is essential that the worst-case execution time (WCET) of such tasks is known in order to ensure that the system works correctly. However, determining the worst-case execution time is a challenge. Simply measuring the execution time of a task for a given input is typically not safe. It is mostly impossible to prove that the conditions leading to the maximum execution time have been taken into account. 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.

Several industrial timing tools and academic prototypes are described and discussed in Wilhelm et al. (2007). Here we concentrate on **AbsInt**'s WCET analyzer **aiT**, which was used successfully at **Hispano-Suiza** to analyze time-critical software. The **aiT** tool statically analyzes a task's intrinsic cache and pipeline behavior based on formal cache and pipeline models, which leads to correct and tight upper bounds for the worst-case execution time. More precisely, **aiT** requires as input the executable to be analyzed, user annotations like targets of indirect function calls and upper bounds on loop iteration counts, a description of the memories and buses, i.e., a list of memory areas with minimum and maximum access times, and the start address of the task to be analyzed. User annotations are only necessary if the required information cannot be detected automatically by **aiT**. They may appear in a separate parameter file called AIS file, or as special comments in the C source.

aiT determines the WCET of the given task in several phases (Ferdinand et al., 2001) (see Figure 1). In the first step a *decoder* reads the executable and reconstructs the control flow (Theiling, 2000). Then, *value analysis* determines lower and upper bounds for the values in the processor registers for every program point and execution context (see section 2), which lead to bounds for the addresses of memory accesses (important for cache analysis and if

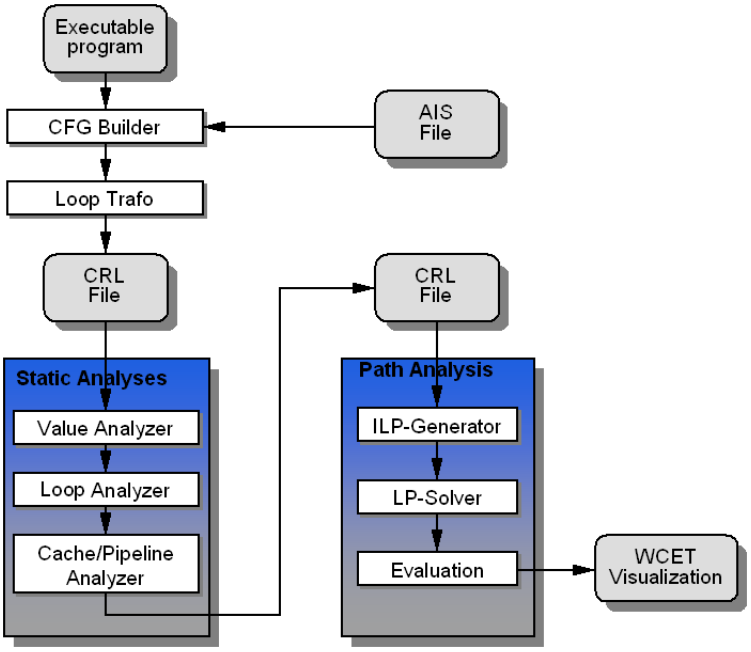

FIG. 1 – *Phases of WCET computation.*

memory areas with different access times exist). 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. Thus value analysis can detect and mark some unreachable code.

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* (Ferdinand et al., 2007), but succeeds in doing so for simple loops only. Bounds for the remaining loops must be provided as specifications in the AIS file or annotations in the C source.

In the general analysis framework of **AbsInt**, an optional *cache analysis* follows, which classifies the accesses to main memory into hits, misses, or accesses of unknown nature. Since MPC555 processors are not equipped with a cache, cache analysis is not included in **aiT** for MPC555, which was used to analyze **Hispano-Suiza**'s applications.

Pipeline analysis models the pipeline behavior to determine execution times for sequential flows (basic blocks) of instructions as done in Schneider and Ferdinand (1999). 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 for each basic block in each distinguished execution context.

Using this information, *path analysis* determines a safe estimate of the WCET. The program's control flow is modeled by an integer linear program (Li and Malik, 1995; Theiling and Ferdinand, 1998) so that the solution to the objective function is the predicted worst-case execution time for the input program.

aiT's results are documented in a report file and as annotations in the control-flow graph that can be visualized using **AbsInt**'s graph viewer **aiSee**. This viewer supports the visualization of the worst-case program path and the interactive inspection of all pipeline and cache states at arbitrary program points.

5 Characteristics of the Analyzed Application

StackAnalyzer and **aiT** were used at **Hispano-Suiza** to analyze a safety-critical application running on Motorola PowerPC MPC555. This application consists of two different and complementary executables corresponding to the application software (AS) and the operating system (OS). The application software was automatically generated with SCADE$^{\mathrm{TM}}$ KCG from Esterel Technologies and implements the main functionality of the system. The operating system essentially is manual code providing the main interface between hardware and application software. The whole system (AS+OS) represents about 28 300 lines of C code (6 738 for the OS and 21 562 for the AS – comments excluded).

Both executables are loaded into external flash memory accessible in burst mode, which ensures high-level performance for the communication with the processing unit. The internal RAM of the processor is used only for holding the stack. Several external RAM components with different access times are added on different chip selects. The entire application runs in supervisor mode. There are no interrupts except for interrupts from the Periodic Interrupt Timer PIT, which are used for periodic scheduling. In particular, there are no interrupts for communication protocols. The Time Processor Units (TPUs) are not used. The software performs a few DMA accesses, which cannot be analyzed by **aiT**; their time must be added to **aiT**'s results.

6 Results of Stack Usage Analysis

StackAnalyzer works on executable files in elf-format. A human operator has to set up a so-called project file specifying the name of the executable, the entry points of the analysis, some annotations providing information necessary for a successful analysis (limits on the call depths of recursive functions, stack usage of external functions, ...), and some more annotations describing the exact context of use of the analyzer. The latter can be used to instruct the analyzer to examine the stack usage of a specific mode of the analyzed software. In our case, there are two modes: an initialization mode and an operational mode.

A **StackAnalyzer** project file was set up for each of the two modes and each of the two executables (AS+OS). The results of the AS analysis were transformed by a Perl script into annotations for the OS analysis. These annotations were needed to inform **StackAnalyzer** about the stack usage of the AS functions called in the OS. Thanks to the Perl script and the batch mode possibilities of **StackAnalyzer**, the process of analyzing **Hispano-Suiza**'s application has been fully automated. Table 1 shows the analysis results in comparison with the project resource allocation specified by the system developers. A comparison with the results of measurements showed an overestimation of less than 5 %.

	AS	OS	AS+OS
Analysis results for initialization mode	144	328	472
Analysis results for operational mode	1984	256	2240
Overall project allocation	2000	400	4000

TAB. 1 – *Stack sizes.*

Frame	Measured time (μs)			Analyzed	Over-
	min	avg	max	time (μs)	estimation
0	3673	3688	3760	4502	19.73 %
1	3741	3741	3760	4458	18.56 %
2	3762	3763	3771	4368	15.83 %
3	3740	3741	3750	4458	18.88 %
4	3667	3701	3722	4503	20.98 %
5	3745	3746	3755	4458	18.72 %
6	2776	3437	3442	4207	22.23 %
7	3746	3747	3747	4458	18.98 %
8	3669	3673	3690	4503	22.03 %
9	3741	3742	3742	4458	19.13 %
				Average:	19.51 %

TAB. 2 – *Application software analysis results.*

7 Results from Using aiT at Hispano-Suiza

7.1 Analysis of the Application Software

In operational mode, the application software is driven by a periodic scheduler dividing time into identical major cycles. Each of these major cycles is divided into 10 minor cycles or time frames in which different routines are called. The routines are enabled or disabled by so-called *condacts* in the SCADE$^{\mathrm{TM}}$ code. If the user does not add some annotations to explain the system behavior, the worst-case path selected by **aiT** will contain almost all of the routines and the resulting WCET would be overestimated. In particular, some annotations must be added to specify the number of the time frame to be analyzed. Apart from this, the automatically generated application code is well understood by the analyzer thanks to its regular structured form so that few more annotations are needed. In particular, code generated by the SCADE 5 code generator does not contain loops, which removes the need for loop bound annotations.

After writing the necessary annotations, the batch mode of **aiT** was used to automatically run the analysis for the 10 time frames. Completing the full analysis required less than 45 minutes with a 2.8 GHz Celeron with 512 MB RAM. Table 2 compares the analysis results with measurements done with HP Agilent on an MPC555 target. Each frame was measured many times. The table shows the minimum, average, and maximum result for each frame. The WCET bound computed by **aiT** was compared with the maximum measurement result.

The average difference between maximum measured time and analyzed time is about 19%. Possible reasons may be the inclusion of control-flow paths that can never be executed in reality, imprecise knowledge of the addresses of memory accesses in presence of memory areas with different access times, and an imprecise modelling of the bus protocol. The measurements might also be too optimistic because they may miss the worst case. Therefore the difference between the real WCET and the WCET estimation computed by **aiT** may be less than 19%.

The overestimation of the WCET could have been reduced if we had looked at every condact structure of the system in order to fit exactly the behavior of the real system. This task has been only done for the "root" functions of the system; in this case, **aiT** was informed about mutually exclusive functions by means of flow annotations.

7.2 Analysis of the Operating System

The manual code of the operating system has a different behavior in the analyzer. The code is built from re-use libraries and contains some loops whose iteration bounds are not automatically found by the analyzer. In the analyzed executable, 62 loops had to be examined manually in order to specify the loop bounds. These loops could be partitioned in a few classes consisting of similar loops with analogous annotations so that the overall annotation effort was moderate after some practice.

aiT's precision was evaluated at some part of the operating system including the OS tasks executed at the beginning of each frame without being called from any AS task, and some OS services such as ARINC reset orders. The measurement was performed by measuring the OS CPU time in absence of any OS requests from the application software, and by measuring the times of most OS services offered to the AS. A few OS services were not included in the measurement. The CPU time used for OS failure (memorization in NOVRAM) was not taken into account because it is dominated by AS failure memorization, and OS and AS failure memorization exclude each other. Periodic backup was also excluded since it is mutually exclusive with the handling of requests from the AS, which takes more time. The actual measurement is initiated by calling the OS scheduler from the external interrupt vector. The time required by the interrupt management and the scheduler call is not taken into account in the measurement.

The overestimation observed for the OS software is higher than the one for AS software: The measured execution time was 855 μs, while **aiT** calculated 1322 μs, which means an overestimation of 55 %. The reason may be the complex control-flow structure of the OS code with more possibilities for control-flow paths that are never executed in reality.

As in stack analysis, some Perl scripts were used to combine AS and OS analysis. This process may become simpler by using a new version of **aiT** that can analyze several executables together (recall that AS and OS software reside in different executables).

8 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.

AbsInt's tools **StackAnalyzer** and **aiT** (timing analyzer) have been used at **Hispano-Suiza** to analyze applications running on a Motorola PowerPC MPC555. Both tools turned out to be well-suited for analyzing the safety-critical applications developed at **Hispano-Suiza**. They can be used either during the validation phase for acceptance tests, or during the development phase providing information about stack usage and runtime behavior well in advance of any run of the analyzed application.

References

Cousot, P. and R. Cousot (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, Los Angeles, California.

Ferdinand, C., R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm (2001). 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*, pp. 469–485. Springer-Verlag.

Ferdinand, C., F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann (2007). New developments in WCET analysis. In T. Reps, M. Sagiv, and J. Bauer (Eds.), *Program Analysis and Compilation, Theory and Practice*, Volume 4444 of *Lecture Notes in Computer Science*, pp. 12–52. Springer-Verlag.

Janz, W. (2003). Das OSEK Echtzeitbetriebssystem, Stackverwaltung und statische Stackbedarfsanalyse. In *Embedded World*, Nuremberg, Germany.

Li, Y.-T. S. and S. Malik (1995). Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*.

Schneider, J. and C. Ferdinand (1999). 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, pp. 35–44.

Theiling, H. (2000). 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.

Theiling, H. and C. Ferdinand (1998). Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, pp. 144–153.

Wilhelm, R., J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström (2007). The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems 5*, 1–47.

Acknowledgement

The authors would like to thank Mr. Vincent Lacroix for his valuable contribution in the frame of his internship at **Hispano-Suiza**.