View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Dagstuhl Research Online Publication Server

Static analysis of WCET in a satellite software subsystem*

Jorge Garrido, Juan Zamorano, and Juan A. de la Puente

Real-Time Systems group (STRAST)
Universidad Politécnica de Madrid (UPM), Spain
str@dit.upm.es

Abstract

This paper describes the authors' experience with static analysis of both WCET and stack usage of a satellite on-board software subsystem. The work is a continuation of a previous case study that used a dynamic WCET analysis tool on an earlier version of the same software system. In particular, the AbsInt aiT tool has been evaluated by analysing both C and Ada code generated by Simulink within the UPMSat-2 project. Some aspects of the aiT tool, specifically those dealing with SPARC register windows, are compared to another static analysis tool, Bound-T. The results of the analysis are discussed, and some conclusions on the use of static WCET analysis tools on the SPARC architecture are commented in the paper.

1998 ACM Subject Classification C.3 Real-time and embedded systems

Keywords and phrases Real-time systems, embedded systems, timing analysis, WCET calculation, static analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2013.87

1 Introduction

UPMSat-2 is a project aimed at developing an experimental micro-satellite that can be used as a technology demonstrator for several research groups at UPM, the Technical University of Madrid.[1]

The software for the mission is being developed by the Real-Time Systems Group at UPM (STRAST)[2], using a model-driven approach [1] that enables auto-generated functional code from different modelling tools, including Simulink,[3] to be included as source code modules. The software structure supporting the concurrency and real-time aspects of the system is also automatically generated from high-level models using a pattern-based approach [3, 12, 11]. The resulting source code is written in Ada 2005 [9] with the Ravenscar profile restrictions [4]. Functional C code is integrated with the Ada code using the standard Ada support for C interfacing.

The hardware platform is based on a LEON computer board [7]. The executable code is generated by means of the GNATforLEON compilation chain [13], which includes the ORK real-time kernel [5].

European software standards for space systems require schedulability analysis to be carried out on on-board real-time systems [6]. The Ravenscar profile enables such analysis

* This work has been partly funded by the Spanish Ministry of Economy and Competitiveness (MINECO), project TIN2011-28567-C03-01 (HI-PARTES).

[1] http://www.idr.upm.es/tec_espacial/upmsat2-eng/01_UPMSAT2.html

[2] www.dit.upm.es/str

[3] http://www.mathworks.com/products/simulink.

© Jorge Garrido, Juan Zamorano, and Juan A. de la Puente;
licensed under Creative Commons License CC-BY

13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza; pp. 87–96

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to be statically performed using well-known response-time analysis methods [10, 2], which require the worst case execution time (WCET) of each task to be known. Since WCET analysis may be complex and the choice of the best approach is not straightforward [15], a case study was chosen in order to assess the suitability of some available tools. The worst-case execution time of one of the UPMSat2 subsystems, ADCS (Attitude Determination and Control Subsystem) was first analysed with RapiTime,[4] a dynamic analysis tool, and the results were presented in a previous paper [8].

This paper describes further work in analysing the ADCS subsystem using a static WCET analysis tool, the aiT tool by AbsInt, which is part of the a^3 analysis toolchain.[5] The results of the analysis are discussed and compared with those obtained with RapiTime.

The paper is organised as follows: Section 2 describes the UPMSat-2 platform, and the relevant architectural characteristics for the rest of the paper. Section 3 describes the SPARC register windows and the way the AbsInt tools deal with them. The approach is compared to that of Bound-T, another static analysis tool. The a^3 tools are described in some detail in section 4. Section 4.2 presents the results obtained from the static analysis, which are compared with those obtained from the dynamic analysis tool. Section 5 presents an alternative approach to modelling the SPARC register window, and its application to three different scenarios. Finally, section 6 presents the conclusions of the analysis.

2 UPMSAT-2 platform

2.1 Computer board

The engineering model used for this study is the same as in [8]. It is based on a GR-XC3S1500 Spartan3 development board[6] with a LEON2 processor[7] at 40 MHz with a 5-stage pipeline and 64 MB of SDRAM.

2.2 SPARC architecture

SPARC (Scalable Processor ARChitecture) is a reduced instruction set architecture (RISC) originally developed by Sun Microsystems [14]. The LEON family of processors is a 32-bit synthesizable VHDL processor core that implements the SPARC V8 architecture. The first LEON processors were originally developed by the European Space Agency in order to provide a stable architecture for European space projects. The aim of the architecture is to provide a platform for which compiler optimization can be easily performed, so that short time-to-market development schedules can be achieved. To this end, the architecture has been simplified with some special characteristics, such as a reduced set of instruction formats, all 32 bits wide and with only two addressing modes, and a separated and configurable floating-point register file. Other relevant aspects of the architecture include a big endian byte ordering, a vectored trap table, as well as separated coprocessor, multiprocessor and floating-point instruction sets.

One of the most distinctive features of the architecture is the use of register windows for handling local storage areas and parameter passing. This feature has a strong influence on the WCET, as discussed in the next section.

[4] http://www.rapitasystems.com/rapitime
[5] http://www.absint.com/a3/
[6] http://www.pender.ch/products_xc3s.shtml
[7] http://www.gaisler.com/leonmain.html

3 SPARC register windows

3.1 Overview

The SPARC architecture defines two types of registers: general-purpose registers and control/status registers. At any given time, 32 registers of 32 bits are provided for the programmer use. They are logically divided into four sets of 8 registers each: *global*, *in*, *local*, and *out* registers. Physically, they are divided into a set of 8 global registers and an implementation-dependent number of 16-register sets. Each 16-register set is in turn logically divided into 8 *in* registers and 8 *local* registers. The remaining 8 registers, the *out* registers, overlap with the *in* registers of the adjacent register set, thus making them accessible from the current set. Each set of 32-bit registers that can be identified in the processor is called a *register window* (see [14] for a detailed description).

The four register sets in a register window are used as follows:

- *Global* registers are shared by all routines.
- *In* registers are used to receive the arguments from the calling routine and return the result at the end of the current routine execution.
- *Local* registers are used to store partial results during the current routine execution.
- *Out* registers used to pass arguments to called functions and receive return values from them.

The aim of this division is to provide a higher number of registers and provide isolation for the local registers in each function. A register window mimics the top of the stack for the currently running routine, thus saving stack access time.

The number of register windows or register sets is implementation-dependent and ranges from 2 to 32. If the nesting level is deep enough, after a sequence of calls, the callee may not have a free register window to use. In that case, a register overflow trap occurs, and the run-time kernel is responsible for providing a new register window to the routine. To this end, a circular buffer of the register windows is commonly used, as shown in figure 1. In order to make space, an implementation-dependent number of register windows are saved in the stack by the overflow trap routine. After that, the call instruction can be resumed and executed successfully. In the same way, it may happen that the previous window is not available when it has to be restored after a chain of routine returns. This situation results in a register window underflow trap, which is handled by the kernel in a similar way.

The SPARC register windows mechanism is useful for reducing the average execution time of a program, but it introduces additional difficulties for computing the WCET of code sections. The reason for it is that register windows introduce a performance dependence on the execution history, in a similar way as cache memories do. The next paragraphs discuss the modelling approach that some static analysis tools currently take in order to deal with this mechanism. An enhanced approach is proposed in section 5.

3.2 Register windows in aiT

The aiT approach to modelling the SPARC register windows is overly simplistic. It assumes that the processor can create an unlimited number of register windows. Consequently, a new window can never overlap the first one, and the tool does not make use of information on window overflows or underflows. Therefore, aiT results are only correct if the code being analysed does not create more windows than available, i.e. if there are no window overflows.

On the other hand, the number of register windows created by the code can be calculated by carrying out a stack analysis with an appropriate tool, e.g. the AbsInt StackAnalizer.

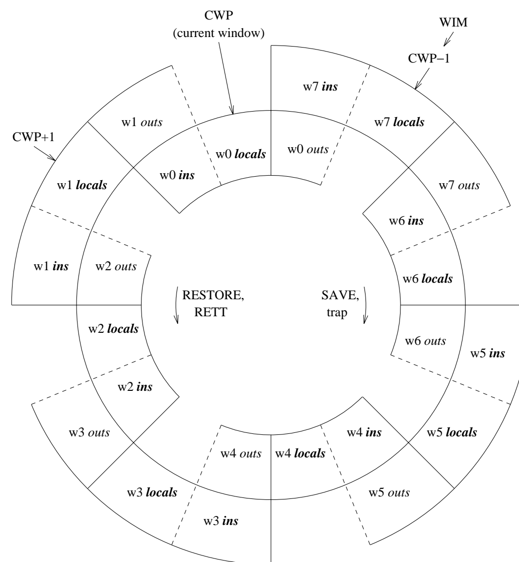

Figure 1 SPARC register window schema. In this example 8 windows are shown, but this is an implementation-dependent value that ranges between 2 and 32. Reproduced from [14].

The tool calculates a range of minimum and maximum register windows used by the code, and this information can be useful to get a better estimation of execution time. However, this is not enough to compute accurate WCET values, since more detail on the total number of windows used, and the times when they are opened or closed, is required for this purpose.

3.3 Comparison with the Bound-T approach

Bound-T[8] is another tool for static analysis of WCET. It provides an upper bound on execution time and stack usage, as well as control flow graphs and call trees, for a variety of processor architectures, including ERC32.[9] Although it does not fully supports LEON processors, it still can be used in a limited way to provide some useful information for this architecture. In particular, Bound-T uses an approach to modelling window registers that is of great interest. It is not only aware of the possibility of an overflow or underflow on each call or restore instruction, but it also offers the possibility to make assertions on the number of register windows that are in use at the entry point of each routine. This feature allows for a better analysis and understanding of the behaviour of the code with respect to register windows. A proposal on how to use Bound-T register window support to complement aiT results is described in section 5.

4 Methodological approach

4.1 Overview of the a^3 toolchain

The aiT tool is integrated in the AbsInt Advanced Analyzer (a^3) toolchain,[10] which also includes tools for stack analysis, value analysis, and other kinds of static analysis. The tools

[8] See `http://www.bound-t.com` and `http://www.tidorum.fi/`
[9] ERC32 is a predecessor of the LEON processor series, based on the SPARC V7 architecture.
[10] `http://www.absint.com/a3`

work on binary executable files, and the results are thus in principle independent of the compiler and source code language. In practice, a wide range of compilers is supported by a^3 for each target.

Static analysis tools are strongly dependent on hardware models, and thus have to be carefully configured in order to get accurate results. Apart from some processor specific features, such as pipeline behaviour, that are already included in the tool, there are other such as FPU, cache and main memory characteristics, which have to be defined by the user.

The a^3 WCET tool (aiT) generates a call graph where the nodes are labelled with their relative execution times, as shown in figure 2. In order to provide a better human interface, the labels are drawn in different colours, depending on whether they lie in the worst-case path or not. Additionally, it outputs a table with detailed information, including self and global WCET, infeasible sections of code, and different context values. An XML output is also provided in order to facilitate further processing of the results by third party tools or scripts.

4.2 ADCS case study

The controller module of the UPMSat2 Attitude Determination and Control System (ADCS) has been used as a case study for static WCET analysis with the a^3 tools. The code is the same that was previously used in an experiment on dynamic analysis carried out with RapiTime [8].

The ADCS controller code has a linear structure, and it mostly consists of arithmetic operations on vectors. In spite of its simplicity, however, the execution times of the floating-point operations is highly variable, and depends on the input values that are read by attitude sensors and processed by the control algorithm.

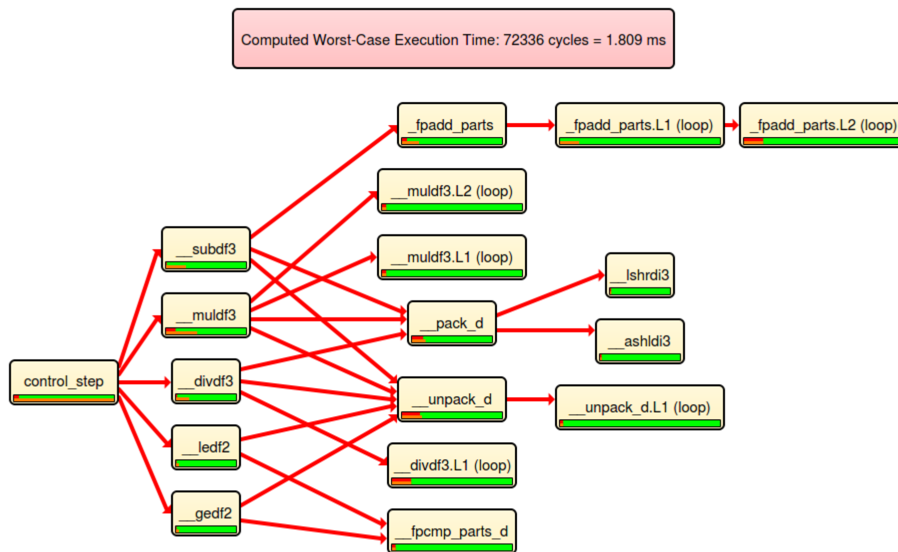

Figure 2 UPMSat-2 ADCS controller worst case execution path graph. As the controller code is linear, all the blocks are in the worst case path.

The results obtained by the a^3 tools for the controller module are shown in figure 2. The estimated WCET value is 72366 cycles, equivalent to 1.809 ms on the target platform. This result is compared with that obtained with RapiTime in table 1. It can be seen that the

Table 1 Comparison between Rapitime and a^3 results.

Tool	WCET (cycles)
Rapitime	8400
a^3	72366
a^3 with real inputs	45000

estimated WCET value is an order of magnitude higher when the static tool, a^3, is used. This could be expected to some extent, taking into account that a^3 computes its results from a model of the computer, whereas RapiTime relays on measurements. However, even considering the different approaches of both tools, the difference between the WCET values seems to be too large.

One of the main reasons for this difference are the different ways that the values of input variables are obtained. In the Rapitime experiment, values of the Earth magnetic field coming from an accurate simulation model are used (see [8] for a description of the testbench). On the contrary, in the a^3 experiment input values are automatically generated within the whole double float range. The actual range of possible values is much more restrictive, but a^3 does not provide a mechanism for imposing such restriction on this data type (although it can be done for other data types). It should be taken into account that the target LEON computer does not have an FPU, and thus the floating-point operations are implemented in software, thus making their execution time highly dependent on the operand values.

In order to compensate for this deviation from the real behaviour, the static analysis experiment has been repeated using a subset of the magnetic field input data values obtained from the simulation model. The values have been included in the model by calling the controller from a wrapper function with the simulation values declared locally. Asserting this function as an entry point, different WCET values are computed for each call. This technique provided a tighter WCET estimation, down to 45000 cycles as shown in the third row of table 1. Notice that this value has been obtained only for comparison purposes, and can not be taken as a real measure of WCET in any case. Moreover, overheads due to register window overflows and underflows have not been taken into account, which surely results in further inaccuracies. An enhanced approach including overflow and underflow overheads is developed in the next section.

5 Modelling register windows

5.1 Register windows and execution time

The fact that a^3 tools do not take into account the register windows overflows and underflows makes the WCET analysis optimistic with respect to this aspect of the SPARC architecture. In order to obtain more accurate WCET estimations, the effect of the overhead incurred by these operations on the execution time has to be appropriately modelled and accounted for in the analysis.

The first step is to get a value for the WCET of the routine that handles register window overflows and underflows, WCET_T. This value can be measured directly on the platform using a hardware monitor such as GRMON.[11] The CPU cycles gap between the first and

[11] `http://www.gaisler.com/index.php/products/debug-tools/grmon`

second call or restore instruction provides such a measurement, including memory accesses. Since the handling routines are linear, there is no cache, and further traps are disabled during their execution, we can safely assume that their execution times are constant.

The run-time kernel can deal with overflows and underflows in different ways. In particular, the number of registers that are saved and restored when a overflow or underflows occurs depends on the implementation, although it has been shown that saving and restoring one window on each overflow/underflow trap is the best approach.

Another important topic is context switches between threads. For instance, some kernels only restore the last window after a context switch. In this way, the context switch time is minimized, but in a concurrent environment each restore can potentially lead to an underflow. Moreover, each save can potentially lead to an overflow if the kernel only provides one valid window to the scheduled thread and keeps the content belonging to the preempted thread in the rest of windows. Therefore, in order to minimize the wort-case number of overflows and underflows, the full content of the register windows should be restored.

In general, the execution time of the overflow/underflow handling routine has to be multiplied by the worst-case number of overflows and underflows in the code, and the result has to be added to the WCET computed by the analysis tool in order to obtain a better estimate of the overall WCET of the code being analysed. The total overhead that can be computed in different situations is discussed in the next paragraphs. Sections 5.2 and 5.3 analyse the cases when only one window is restored or saved by the trap routines, whereas section 5.4 analyses the situation when more windows are restored or saved.

5.2 One window, no full context restore

The stack analysis tool provides the minimum and maximum number of register windows that can be used and the number of overflow/underflow traps that can occur in a single execute of a function f. Let this number be T_f, and let n_f be the number of times f is called in the worst-case path. The total worst-case number of traps occurring in f is thus

$$N_f = n_f \times T_f$$

The total number of traps in the worst-case path is

$$N = \sum_{f \in F} N_f$$

where F is the set of functions.

Finally, the total worst-case overhead is

$$\text{WCOH} = N \times \text{WCET}_T$$

where WCET_T is the worst-case execution time of the trap handler routines.

Applying this technique to the case study, Bound-T reports a total of six functions causing overflows, with a total number of overflow traps $N_O = 25$ and a similar number of underflow traps, $N_U = 25$. The WCET is not the same for overflow and underflow traps, and the respective values are $\text{WCET}_O = 156$ and $\text{WCET}_U = 188$ cycles. Hence the total overhead caused by the overflow and underflow traps is

$$\text{WCOH} = 25 \times 156 + 25 \times 188 = 8600\,\text{cycles}$$

This value is significant, as it amounts to an 11.56 % of the WCET computed by a[3] and a 102.38 % of that computed by Rapitime.

5.3 One window, full context restore

The ORK kernel used in the UPMSat-2 software saves and restores only one window per trap, as previously considered, but in this case the full state of the registers is restored on each context switch. Therefore, the worst-case number of overflows in a code section equals the maximum depth of register windows it can create. This number is usually much lower than that obtained in the previous situation. Underflows can only occur if the depth of windows saved in the stack is greater than the number of physical windows.

For the ADCS case study, both a^3 and Bound-T reported a maximum register window depth $D_W = 3$. This means that the maximum number of overflows is $N_O = 3$. Since the LEON processor has 8 real register windows, no underflow can occur. The overhead is now

$$\text{WCOH} = 3 \times 156 + 0 \times 188 = 468\,\text{cycles}$$

This value is much lower than the previous one. It amounts only to a 0.63 % of the a^3 WCET value and a 5.57 % of the Rapitime value.

5.4 More than one window

The SPARC architecture enables from 2 to 32 register windows, and implementations can restore between 1 and the total number of sets. This greatly increases the complexity of the analysis, as the number of possible behaviour may become very high. A full study of all the implementation possibilities is out of the scope of this paper, but previous versions of the ORK kernel are discussed as an example.

- In ORK 1.0 only the last window was restored after a context switch. This case is the "one window with no restore" case, as any save and restore may cause a trap.
- In later ORK versions, the full register set is left as it was before the context switch. In order to achieve this result, the trap handling routines for register window overflow and underflow save or restore the full set of windows. For example, on a processor with 8 register windows, there are 7 windows available to user software. Therefore, after 7 consecutive saves a window overflow occurs. In the worst case, user code may save and restore windows in the edge of the window invalid mask,[12] generating a trap on each save or restore.

In the case study the worst case happens when the user function is called using the last valid window, and thus every procedure call causes a window overflow. This includes all calls to floating-point arithmetic routines, which are frequent. Similarly, every return from the routines causes an underflow. Therefore the overhead is just one overflow and underflow less than the "one window, no restore" case. i.e.

$$\text{WCOH} = 24 \times 156 + 24 \times 188 = 8256\,\text{cycles}$$

Since more than one window is saved or restored on each overflow or underflow trap, the execution time of the trap handling routines may increase. Therefore, the above values should be taken as a lower bound.

In all cases, the computed overhead must be added to the WCET computed with the the basic analysis of section 4.2. Table 2 summarizes the results compared to the basic analysis.

[12] The window invalid mask is a bit map of valid windows [14].

Table 2 Rapitime and a^3 results with register windows overhead (times in cycles).

Tool	Basic analysis	1 w. full restore	1 w. no restore	7 w. full restore
Rapitime	8400	8868 (+5.57%)	17000 (+102.38%)	16656 (+98.28%)
a^3	72366	72834 (+0.63%)	80966 (+11.56%)	80622 (+11.28%)

6 Conclusions

Modelling modern processors is a complex task, due to the wide range of hardware acceleration features they include, such as cache memories, memory management units, coprocessors, or multicore architectures. Although the SPARC v8 is far from new, modelling its set of register windows is not trivial and indeed complicates the task of estimating execution times on this kind of processors. In general, it can be said that the SPARC registers architecture improves efficiency and reduces the overall execution time of the application code. However, if not properly analysed, it can lead to imprecise, unsafe results.

Some software elements can also have an influence on the execution time of code running on register window architecture. In particular, run-time kernels may deal with window overflows and underflows in different ways. Such differences may result in different levels of overheads on the WCET values, as shown in section 5. Other sources of inaccuracy may arise that make it difficult for analysis tools to predict the execution behaviour of code. For example, loop bounds may depend on input values that are unknown at the time of analysis. A way to cope with this issue is to use assertions to restrict the number of possible execution paths. However, assertions are often hard to establish and prone to errors, and thus unsafe.

The scenarios analysed in section 5 and summarized in table 2 show that different hardware and software implementations of the SPARC register windows produce different levels of overhead in the case execution time that must be taken into account for the estimation of WCET.

Acknowledgements

The authors would like to thank AbsInt and Tidorum for their active collaboration and the support provided. We would especially like to express our gratitude to Christian Hümbert from AbsInt, Enrico Mezzetti from Università degli Studi di Padova, and Niklas Holsti from Tidorum, for their support and personal implication.

References

1 Alejandro Alonso, Emilio Salazar, and Juan A. de la Puente. Design of on-board software for an experimental satellite. In *Jornadas de Tiempo Real — JTR-2013*, 2103.

2 Neil Audsley, Alan Burns, Rob Davis, Ken Tindell, and Andy J. Wellings. Fixed priority preemptive scheduling: An historical perspective. *Real-Time Systems*, 8(3):173–198, 1995.

3 Matteo Bordin and Tullio Vardanega. Automated model-based generation of Ravenscar-compliant source code. In *Proc. 17th Euromicro Conference on Real-Time System*, ECRTS'05, pages 59–67, Washington, DC, USA, 2005. IEEE Computer Society.

4 Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the Ada Ravenscar profile in high integrity systems. *Ada Letters*, XXIV:1–74, June 2004.

5 Juan A. de la Puente, José F. Ruiz, and Juan Zamorano. An open Ravenscar real-time kernel for GNAT. In Hubert B. Keller and Erhard Plödereder, editors, *Reliable Software*

Technologies — Ada-Europe 2000, number 1845 in LNCS, pages 5–15. Springer-Verlag, 2000.

6 European Cooperation for Space Standardization. *ECSS-E-ST-40C Space engineering — Software*, March 2009. Available from ESA.

7 Gaisler Research. *LEON3 – High-performance SPARC V8 32-bit Processor. GRLIB IP Core User's Manual*, 2012.

8 Jorge Garrido, Daniel Brosnan, Juan A. de la Puente, Alejandro Alonso, and Juan Zamorano. Analysis of WCET in an experimental satellite software development. In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 81–90. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2012.

9 ISO. *Ada Reference Manual ISO/IEC 8652:1995(E)/TC1(2000)/AMD1(2007)*, 2007. Available at `http://www.adaic.com/standards/ada05.html`.

10 Mathai Joseph and Paritosh K. Pandya. Finding response times in real-time systems. *BCS Computer Journal*, 29(5):390–395, 1986.

11 Enrico Mezzetti, Marco M.Panunzio, and TullioVardanega. Preservation of timing properties with the Ada Ravenscar profile. In Jorge Real and Tullio Vardanega, editors, *Reliable Software Technologies – Ada-Europe 2010*, number 6106 in LNCS, pages 153–166. Springer-Verlag, 2010.

12 José Pulido, Juan A. de la Puente, Matteo Bordin, Tullio Vardanega, and Jérôme Hugues. Ada 2005 code patterns for metamodel-based code generation. *Ada Letters*, XXVII(2):53–58, August 2007. Proceedings of the 13th International Ada Real-Time Workshop (IRTAW13).

13 José F. Ruiz. GNAT Pro for on-board mission-critical space applications. In Tullio Vardanega and Andy Wellings, editors, *Reliable Software Technologies – Ada-Europe 2005*, volume 3555 of *LNCS*. Springer-Verlag, 2005.

14 SPARC International, Upper Saddle River, NJ, USA. *The SPARC architecture manual: Version 8*, 1992.

15 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.