PapaBench : A Free Real-Time Benchmark

Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun and Marianne De Michiel
IRIT - University of Paul Sabatier
F-31062 Toulouse France
{nemer, casse, sainrat, bahsoun, michiel}@irit.fr

Abstract

This paper presents PapaBench, a free real-time benchmark and compares it with the existing benchmark suites. It is designed to be valuable for experimental works in WCET computation and may be also useful for scheduling analysis. This benchmark is based on the Paparazzi project that represents a real-time application, developed to be embedded on different Unmanned Aerial Vehicles (UAV).

In this paper, we explain the transformation process of Paparazzi applied to obtain the PapaBench. We provide a high level AADL model, which reflects the behavior of each system component and their interactions.

As the source project Paparazzi, PapaBench is delivered under the GNU license and is freely available to all researchers. Unlike other usual benchmarks widely used for WCET computation, this one is based on a real and complete real-time embedded application.

1. Introduction

When designing a real-time system, it is mandatory to have a predictable timing of the system. While underestimating the execution time of tasks may cause catastrophic disasters especially in critical hard real-time systems, overestimating the execution time may also cause an oversizing of the running hardware.

To prove these timing constraints, it is essential to know the Worst Case Execution Time (WCET) of a program running on a particular hardware system. The real-time system designers use it to check the timing deadlines satisfaction of the tasks while many real-time operating systems rely on this information to perform scheduling. Moreover, in embedded system design, the WCET of the software is often required in order to decide how to partition hardware / software.

As any piece of software, the WCET computation needs to be experimented, evaluated and compared. To achieve this goal, this paper introduces PapaBench, a real time benchmark, describing a complete embedded system driving a UAV. Designed to be a valuable base for experimental work in the WCET computation by static [1, 2, 3, 4, 5] or dynamic [6] analyses, it may be also very useful for scheduling analysis of applications since it provides concrete tasks and interrupts with their timing constraints and precedence rules. This benchmark will make experimental results more realistic than existing WCET benchmarks [7, 8] since the tasks encountered are close to those running in real avionic systems.

The rest of this paper is organized as follows. Section 2 provides a complete description of Paparazzi. Section 3 presents our PapaBench model in AADL [9, 10], which maps the Paparazzi C sources into a list of tasks and interrupts. Section 4 describes the PapaBench genesis, the adaptation to compile this benchmark on different architecture and the mapping of the application sources with the AADL model. We compare our benchmark with existing real-time benchmarks in section 5 and section 6 concludes this paper.

2. The Paparazzi Project

The "Paparazzi" project, created in 2003 by P. Brisset and A. Drouin [11, 12], is an attempt to build a cheap fixed-wing autonomous UAV executing a predefined mission. It develops a complete system hardware and software that may be installed on a variety of aircrafts. Such a system has limited autonomies, a 2-5 kg total aircraft weight, a 25 km maximum flight distance, a one hour flight duration, a 50 km/h maximum speed and a 500 g maximum payload.

It comprises an embedded system and a ground station as shown in Figure 1. The embedded system

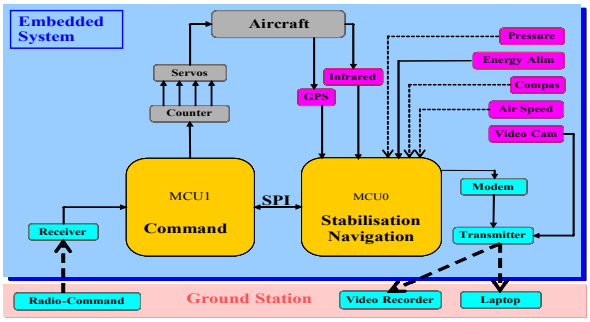


***Figure 1**: Paparazzi*

consists of a control card, a GPS receiver (μblox SAM-LS with 16 channels), a two-axis differential infrared sensor, a radio transmitter and servo-commands controlling gaz and wings. We have also a list of devices supplying voltage, pressure, heading and so on.

The control card is designed as a bi-processor architecture, separating the radio / servo commands management from the autopilot task, holding two RISC, ATMEL AVR micro-controllers [13]. MCU1 (ATMega8, called Fly-By-Wire) features a 16 Mhz / 16 MIPS processor with 1 Kb SRAM, a 8 Kb flash memory and 512 bytes EEPROM, that manages radio-command orders and servo-commands. MCU0 (ATMega128, called Autopilot) provides a 16 MHz / 16 MIPS processor with

ECRTS 2006
6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2006/678

4 Kb SRAM, 128 Kb flash memory and 4 Kb EEPROM. It runs the navigation and stabilization tasks of the aircraft. The micro-controllers are inter-connected by a SPI serial link in a master (MCU0) / slave (MCU1) mode.

The ground station consists of a usual radio-command, a radio receiver and a laptop. The audio and video channels of the receiver are respectively connected to the laptop and to a video tape recorder. The laptop receives information about the running mission while a variety of interfaces visualize the flight parameters, the flight path and all the messages held by the aircraft.

Although the ground station is largely developed, we are only interested in the embedded system that constitutes the core of our benchmark. Indeed, the ground station software does not exhibit any hard real-time code.

The embedded system has two basic operation modes: "manual" mode and "automatic" mode. In "manual" mode, MCU1 receives the radio-command instructions from the ground station and dispatches them to MCU0. MCU0 analyses this information, performs the stabilization and returns the flight commands to the MCU1 that transmits them to the servos.

On the other hand, in "automatic" mode, MCU0 manages the aircraft navigation using the GPS and the infrared sensor while MCU1 only receives the flight commands and transmits them to the servos. In this mode, the aircraft has a specific mission defined in a high level language. Thereby, there are three control levels: the mission, the navigation and the stabilization.

If MCU0, possibly crashed, sends no more commands and the radio-command is unreachable, the system switches to the failsafe mode: the engines are stopped and the aircraft glides to the ground.

In conclusion, Paparazzi is a realistic real-time embedded system exhibiting a quite complex behavior. Yet, unlike most equivalent industrial systems, the sources are freely available.

3. Modeling with AADL

Unlike other WCET benchmarks, PapaBench is close to actual running systems, rendering the experimentation results more realistic and making it possible also to handle real effects of task chaining. In order to cope with the Paparazzi embedded system, we have first produced an AADL model describing the whole system. Unlike other benchmarks, PapaBench is not a collection of independent programs but provides a full application. Consequently, we need a way to split it in tasks and to model the dynamic behavior of the system: AADL is widely used in avionics field to achieve this goal.

3.1. About AADL

We have decided to depict this application in AADL (Architecture Analysis and Design Language) because it is a formal specification of real-time embedded, fault-tolerant, securely-partitioned, dynamically-configurable systems. It covers the domain of distributed multiple-processor hardware architectures as found in avionics, robotics and automotive.

A system modelled in AADL consists of an application software mapped to an execution platform. It describes how components are combined into subsystems, how they interact and how they are allocated to hardware components.

AADL has ten basic component's types divided into three categories: software, hardware and composite. Data, thread, thread group, process and subprogram constitute the first category. The hardware category holds processors, memories, buses and devices. The “system” is the only composite element.

3.2. AADL Usage

An AADL model depicts the overall application with an accurate model of the whole embedded system. As AADL provides a textual and graphical view of the system, the user can easily understand the internal application work.

Moreover, the well-defined AADL language and its openness may be used to perform automatic processing. For example, different schedules may be generated from the description: in our experimentation, we plan to use CHEDDAR [14] for this task. In particular, we plan to use the scheduling results and the AADL model of the application to analyse the WCET of a whole application running cycle. This analysis may be used to evaluate the full system workload or to handle hardware dependencies between tasks in order to improve the WCET accuracy.

Finally, although the model is based on a real application and while we do not perform functional simulation, we have some freedom to change it according to our experimentation needs. We may add / delete components, change components properties and / or add new properties to evaluate application parameters. As we are especially interested in timing constraints in WCET and scheduling analysis, an AADL model can be very useful to evaluate these properties without having to change the application structure.

3.3. Paparazzi AADL Model

We found the different control levels and the corresponding timing constraints in the report on the Paparazzi project [11].

Based on this reference, and after analyzing the C files of the embedded software, we have identified a list of tasks executed in this application as well as their timing constraints and their precedence rules. We have also determined the interrupts used to drive the hardware. Table 1 on next page shows the tasks and the interrupts executed by MCU1. Table 2 displays those treated by MCU0. We provide for each task and interrupt an identifier used in the following section, a description and the appropriate frequency.

The precedence rules that sets an order on tasks execution are depicted as a graph in Figure 2. This order is required by data dependencies (edges marked with 1) or control dependencies (edges marked with 2). The dashed arrows reflect the precedence rules valid in manual mode, the plain arrows represent the precedence rules in automatic mode and the thick ones are valid in both modes. The white circles reveal tasks executed only in manual mode, the gray circles are tasks executed in automatic mode and the dark ones are executed in both

ECRTS 2006
6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2006/678

ID	Description	Frequency
T1	Receive Radio-Command orders	40Hz
T2	Send Data to MCU0	40Hz
T3	Receive MCU0 values	20Hz
T4	Transmit Servos	20Hz
T5	Check Failsafe	20Hz
I1	Transmission Servos interrupt	-
I2	SPI interrupt of MCU1	-
I3	Radio interrupt	-

Table 1: *MCU1 tasks and interrupts*

ID	Description	Frequency
T6	Managing Radio orders	40Hz
T7	Stabilization	20Hz
T8	Send Data to MCU1	20Hz
T9	Receive GPS Data	4Hz
T10	Navigation	4Hz
T11	Altitude Control	4Hz
T12	Climb Control	4Hz
T13	Reporting Task	10Hz
I4	SPI interrupt of MCU0	-
I5	Modem interrupt	-
I6	GPS interrupt	-

Table 2: *MCU0 tasks and interrupts*

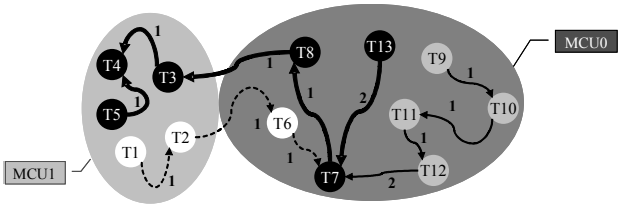


Figure 2*: Precedence Rules Graph*

modes.

In manual mode, MCU1 receives information from the radio command (T1) and transmit it to MCU0 (T2). MCU0 executes T6, T7 and T8 to analyze radio-command instructions, to perform stabilization and to return the flight commands. Then T3 and T4 occurs to receive data from MCU0 and send it to servos. T4 enables the interrupt I1 to send information to the servos. This scenario persists as long as the system stays in this mode.

The automatic mode is activated by a radio command order or when the radio command is no more reachable. AADL provides modes to record the system operational mode. Variation in operational modes is triggered by events.

In automatic mode, T9 analyses the messages held by the GPS. The navigation (T10) is realized at the same frequency as the GPS information delivery, it also controls the mission to consider exceptional events. In this mode T10, T11 and T12 occur always in this order before the stabilization task T7. They cannot execute separately. They are followed by [T7, T8, T3, T4]. Note that the repetition of [T7, T8, T3, T4] is constrained by their period which is less than the period of T10, their execution time and the execution time of the previous tasks : these tasks may be viewed as the feedback control loop that must satisfy the navigation commands. In both modes, MCU0 reports changes in the aircraft path, in the operational mode, in the navigation and so on to the ground station in a task executed at 10 Hertz.

At the hardware level, the Paparazzi system has two subsystems, one for each micro-controller. Each system describes the execution process as a set of threads defining the tasks and interrupts executed in its context, the list of devices and the relations between components.

3.4. Variable Complexity

In AADL, each component type can be characterized by a set of properties. To include the timing constraints in our model, we added for each thread the period property and the dispatch protocol that can be either aperiodic, sporadic or periodic. An aperiodic task occurs at arbitrary times but can be delayed for a limited time, while a sporadic one occurs at irregular intervals with a maximum or minimum period between two consecutive executions. The properties of the components can be changed or extended in order to reflect the user demands.

The inputs / outputs in Paparazzi are managed as aperiodic interrupts. Usually, the avionics software does not support interrupts because the WCET cannot be accurately computed with the current techniques. However, we are not compelled to use the static scheduling provided in the Paparazzi C code. One may consider the AADL model, its tasks (including interrupts) and its matching code in C sources but different scheduling and timing properties may be experimented according to our needs. This leads us to define several models, from the simplest one to the most realistic one.

We can begin to work with the simpler configuration of the system, assuming that all tasks and interrupts are periodic. Then, we may improve our analysis toward more complex configurations, close to the real application behavior. In the AADL model, it is easily achieved by varying the “Dispatch_Protocol” property value in the set: “periodic”, “sporadic” and “aperiodic”.

We can also consider the preemption between tasks. For this purpose, we extended AADL with a new property, called “Preemption”, only applicable to the threads. This property indicates the preemption type that may be one of “System_Preemption”, “Time_Sharing_Preemption” or “Non_Preemptive”. We must also choose a preemptive scheduling protocol for the processor. This new property offers a framework for many kind of studies including WCET computation.

As an example of use of these different levels of complexity, we plan to experiment an approach allowing the WCET computation for a complete cycle of the application. We intend to begin our WCET analysis with the basic level where we will consider periodic tasks and periodic interrupts with no preemption, in order to define a possible schedule with the scheduler.

4. PapaBench Genesis

Source code, schematics and documentation of the Paparazzi project [12] are freely released under the GNU

ECRTS 2006
6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2006/678

license. This section explains our analysis of Paparazzi, the transformation process to obtain PapaBench and the changes required to compile the benchmark.

4.1. System Instantiation and Restriction

The Paparazzi distribution is only available for a Linux environment but may be configured for several aircraft configurations. As we are not interested in the details of the hardware control, PapaBench is only bound to the default aircraft configuration. It includes a MC3030 radio-command, a Twinstar3 model-making aircraft, the flight plan used during the first European MAV Flight Competition held in Braunschweig Germany on July 13 2004 and a classic ground station as described in section 2.

The first generation of the embedded system enables us to save C header files, generated from XML sources. These XML files contain the configuration of the airframe, the radio commands and the flight plan that constitutes the mission. They make the Paparazzi project applicable to many different aircrafts and allow to realize different flight plans. After the generation, we saved the generated header files and included them in the benchmark without having to preserve the XML sources.

Then we analyzed the static sources and the generated headers in order to create PapaBench. We have excluded the sources of the ground station in charge of monitoring the flight, displaying statistics, programming the mission and generating the embedded system sources: they are composed of a mixture of OCAML and Perl programs not really involved in the embedded system.

4.2. Mapping the AADL Model

Using the OTAWA project [15, 16], a framework to experiment WCET computations and binary static analyses, we have developed a program generating the Program Call Graph (PCG) of an executable file and some other statistics about the executed binaries.
The PCG gives a general idea of the complexity of the application and it enables the user to have a clear view of the function call chains without reverting to the sources.
They also provide a map of the tasks identified in the logical analysis of the system to the matching implementation in the C sources.
The PCG of Fly-By-Wire and Autopilot are represented in Figures 3 and 4 respectively. The grey ellipses represent task implementation code. The identifier of this task is marked in a dark label.

In the Autopilot PCG, T10 is course_pid_run and either nav_home if the system is in failsafe mode, or nav_update in automatic mode. The dashed ellipses show interrupts interfering with the execution of the subsystems tasks. One may notice that some subprograms are not part of any task: this code is only called at startup time and is not involved in the system execution during the flight.

4.3. Compilation Details

Our objective was to enable a user to compile the benchmark, without the requirement of the whole Paparazzi building environment, for different architectures. Until now, we have experimented the compilation for PowerPC and x86 architectures using the GCC compiler suite but it should be easy to adapt PapaBench to other configurations.

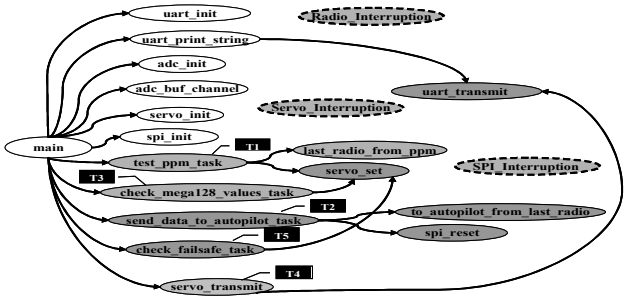


***Figure 3**: Fly-By-Wire PCG*

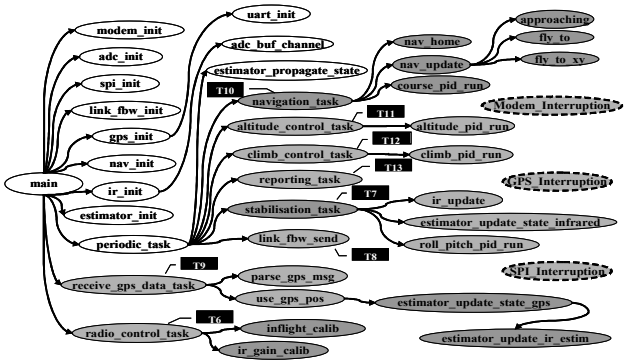


***Figure 4**: Autopilot PCG*

To compile the benchmark, one must extract the archive and edit the default configuration files, in the conf/ directory, to change the top directory path and the compiler command according to the target architecture. A simple call to make in the distribution top directory should compile everything.

It is important to mention that PapaBench includes headers files from the AVR C libc library project containing macros providing access to the AVR hardware registers like IO ports, timers, and so on. As the benchmark does not target hardware simulation, either the hardware registers only matter by their temporal properties, or they may be simply considered as simple memory accesses.

As they are mapped to low addresses (between 0x20 and 0x100), this might be impeding for some platforms where the addresses of interrupts vectors appear at this location. Fortunately, we can get rid of this problem by assigning in the compilation flags a compatible value to the SFR_OFFSET definition, which is the base of the hardware register addresses.

5. Comparison with Other Benchmarks

Benchmarking constitutes a critical part of the design process. As real applications are not easily available to researchers due to the confidentiality criteria surrounding the industrial estate, real-time benchmarks are rare and often disconnected from the surrounding particularities of real-time systems. This section gives an overview of these benchmarks and compare them to PapaBench.

5.1. Other Benchmarks

Real-time benchmarks are usually a collection of basic algorithms found in real-time systems.

MiBench [7], for example is a set of 35 embedded applications divided into six suites, each one targeting a specific area of the embedded market. The six categories are: 1) automotive and industrial control, 2) consumer devices, 3) office automation, 4) networking, 5) security and 6) telecommunications. All the programs are available in standard C source code and are portable to any platform that has compiler support. Some modifications has been made to the source to promote the portability of the benchmark and the extensibility of the data set. Where appropriate, MiBench provides a small and large data set. The small data set represents a light-weight, useful embedded application of the benchmark, while the large data set provides a real-world application. This benchmark has many similarities to the EEMBC suite as described on their website [17] but MiBench is composed of freely available source code. We only compare the category (1) of MiBench suite with PapaBench because other categories are not used in hard real-time systems.

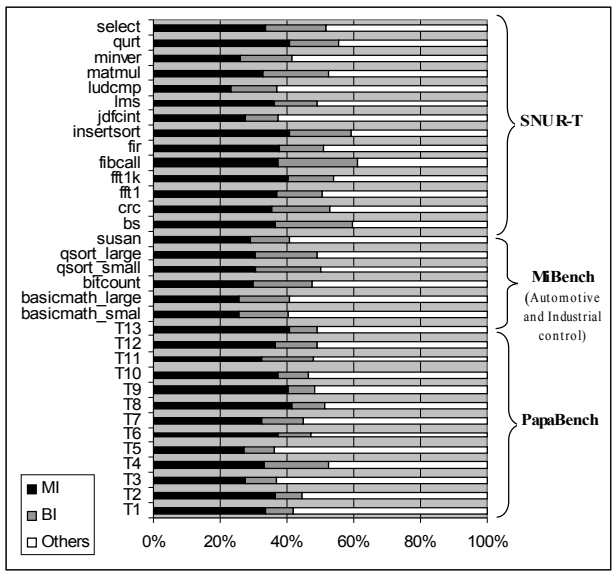
The SNU Real-Time Benchmarks suite [8], consists of C sources collected from numerical calculation programs and DSP algorithms as binary search program, fast Fourier transform, Fibonacci series function, insertion sort, square root calculation, matrix multiplication and many other programs. The benchmarks have the following structural constraints: no unconditional jumps, no exit from loop bodies, no *'switch'* statement, no *'do...while'* construct, no multiple expressions joined by *'or'*, *'and'* operations and no library calls. These restrictions are caused by the limited capabilities of the compiler involved in the experimental analysis environment used by the benchmark creators.

The benchmarks mentioned above are disconnected from the surrounding particularities of real time systems. Their functions found are executed alone out of the context of a real application. On the other hand, PapaBench tasks are embedded in a real system with hard timing constraints. This feature allows the analysis of effects of tasks on the execution of other ones. It is worthy to use such an application because of its similarities with the industrial real-time applications.

5.2. Code Characteristics

We have used the OTAWA framework to characterize the PapaBench code as well as MiBench and SNU R-T codes. Tables 3 gives, for each benchmark, the branching rate, the memory access rate, the average size and the maximum size of basic blocks. Figure 5 provides, for each task of the benchmarks, the rate of memory accesses (black area), of branching instructions (gray area) and of other instructions.

First, we can see that PapaBench has small basic blocks except for T1 and T9 where the maximum size of basic blocks is 137 and 110 respectively: it seems that the radio management and GPS data analysis require a lot of computations. SNU RT functions also have small basic blocks except for *jdfcint* and MiBench_Automotive has big basic blocks for the majority of its tasks : MiBench is too much oriented toward computations unlike the other benchmarks.



MI(Memory instruction rate), BI (Branching Instructions)

Figure 5: *Instruction repartition*

	R_B	R_M	AS_{BB}	MS_{BB}
PapaBench	0,093	0,383	7,06	137
MiBench ***(Automotive & industrial control)***	0,181	0,275	4,63	150
SNU R-T	0,15	0,341	5,27	89

R_B (Branching rate), R_M (Memory access rate), AS_{BB} & MS_{BB} (average & maximum size of Basic Blocks)

Table 3*: Statistics*

We found a high level rate of memory accesses in the three benchmarks which reflects the importance of the memory hierarchy analyses in the WCET computation. While, in case of PapaBench, it is caused by lots of hardware register accesses, other benchmarks seem to have a too big memory foot print.

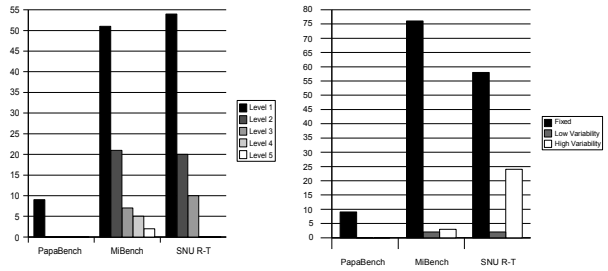
To sum up, PapaBench has some similarities with other real-time benchmarks as they all have close memory access and branching rates. This also confirms that these benchmarks are close to real applications. Next section will show that differences exist and these statistics are not enough to characterize a benchmark.

5.3. Loop Complexity

CFG and syntax tree representations are not enough for static WCET computation since they don't identify bounded execution paths. Hence these representations have to be completed by some information to restrain the number of executable paths to consider in the analysis. In this paragraph, we discuss the PapaBench loop complexity and compares it with the other benchmarks.

Graph (a) of Figure 6 displays for each benchmark, the loops repartition among 5 nested levels. The darker column represents top level loops counts and the columns get brighter as the level is deeper. Moreover, graph (b) reflects the variability level of loops maximum iteration numbers. We distributed benchmarks loops over three levels: 1) for loops with fixed iteration number, 2) for loops with little variation in the iteration number (depending on parameters with a constant during the

ECRTS 2006
6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2006/678



(a) Loops Depth *(b) Loop Bound Variability*

***Figure 6**: Loop Complexity Analysis*

function call) and 3) for loops with high variability degree (induced by loops nesting with an inner loop bound depending on the outer loop induction variable).

The loops encountered in PapaBench are mostly *for* loops, we have only two *while* statements. The *for* loops maximum iteration number is fixed but the *while* loops analyses gives a maximum iteration number of 0 or 1. Thus, we mostly do not have variations in loop bounds. Moreover, we can notice that PapaBench loops are simple with no nesting as shown in graph (a). On the other hand, MiBench and SNU-RT benchmarks contain different nesting levels of loops with variable iteration numbers. A high level of variability makes WCET analyses more complicated or increases the approximation pessimism since the user have to provide an upper bound of loops iterations. If the loop bound is not represented in a fine way (as constants for example), the real number of iteration may be over-estimated due to loops nesting and the variability of loops bounds. However, the PapaBench case and our experience in avionics software show that we have more often simple loops with a fixed number of iterations. This makes WCET calculation accurate and closer to the real WCET. In the other hand, it seems that other benchmarks exhibit over-complicated program structures.

6. Conclusion

Benchmarking is a critical problem in WCET computation because real applications are not easily available due to the confidentiality criteria surrounding the industrial estate. In this paper, we introduced PapaBench, a complete real-time embedded application derived from a real application used to control a UAV. This prominent feature makes it mostly useful in WCET and scheduling analyses and unique among the other existing benchmarks. We have given a whole description from the system point of view, using an AADL model, and an instruction level analysis. We have also compared it with existing real-time benchmarks to denote similarities and advantages that makes it useful and unique in WCET computation domain.

In the near future, we plan, to perform new analyses of the PapaBench AADL model: we will consider two levels of complexity, for the periodicity of tasks and interrupts. These restrictions will be used to validate a WCET computation approach based on the whole application cycle. Note that PapaBench sources and AADL model are available at http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=22.

7. References

[1] Y.-T. S. Li, S. Malik. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. 16th IEEE Real-Time Systems Symposium, pages 298-307, December 1995.

[2] Y.-T. S. Li, S. Malik. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. 17th IEEE Real-Time Systems Symposium, 1996.

[3] C. Ferdinand et al. Applying Compiler Techniques to Cache Behavior Prediction. ACM SIGPLAN Workshop on Languages, Compilers, and Tools Support for Real-Time Systems: 37–46.

[4] F. Muller. Generalizing Timing Predictions to Set-Associative Caches. Technical Report TR 96-66, Institut für Informatik, Humboldt-University, July 1996.

[5] Y. Tan, V. Mooney. Integrated Intra- and Inter-Task Cache Analysis for Preemptive Multi-Tasking Real-Time Systems. 8th International Workshop, SCOPES 2004, in Lecture Notes on Computer Science, LNCS3199, pages 182–199, 2004.

[6] I.Wenzel, B.Rieder, R. Kirner, P. Puschner. Automatic Timing Model Generation by CFG Partitioning and Model Checking. Design, Automation and Test in Europe, Volume 1, pages 606-611. March 2005.

[7] M.R. Guthaus, J.S. Ringenberg. Austin, T. Mudge and R.B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. 4th Workshop on Workload Characterization, Dec. 2001, Austin, TX.

[8] SNU Real-Time Benchmark Suite. http://archi.snu.ac.kr/realtime/benchmark.

[9] P. Feiler, D. P. Glush, J. J. Hudak, B. A. Lewis. Embedded System Architecture Analysis Using SAE AADL, June 2004.

[10] SAE International. Architecture Analysis & Design Language (AADL), August 2004.

[11] P. Brisset. Drones civils perspectives et réalités. Technical report, Ecole nationale de l'aviation civile, August 2004.

[12] www.recherche.enac.fr/paparazzi

[13] ATMEL Corporation. ATMega128 complete datasheet. http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

[14] F. Singhoff, J. Legrand, L. Nana. AADL resource requirements analysis with Cheddar. LYSIC/EA 3883.

[15] H. Cassé, C. Rochange, P. Sainrat. An open Framework for WCET Analysis. IEEE Real-Time Systems Symposium-WIP session, pages 13-16, Lisbon, December 2004.

[16] H. Cassé, C. Rochange, P. Sainrat. OTAWA, a framework for experimenting WCET computations. 3rd European Congress on Embedded Real-Time, Toulouse, December 2005.

[17] EEMBC Real-Time benchmarks . http://www.eembc.com.